Trabajo Final de Grado

Grado en Ingeniería en Tecnologías Industriales

Implementación del algoritmo AES en un microcontrolador PIC18F

MEMORIA

Autor: Laura Fabregat Francés
Director: Manuel Moreno Eguílaz

Convocatoria: junio 2017

Escuela Técnica Superior de Ingeniería Industrial de Barcelona

Resumen

Actualmente, la criptografía tiene un importante papel en la vida cotidiana. Es la encargada de mantener la privacidad y la confidencialidad en los diferentes tipos de comunicaciones: correos electrónicos, compras, banca electrónica, contrataciones de servicios por internet, etc.

El campo de uso de la criptografía se extiende hasta los entornos industriales, donde la exposición a ataques en las diferentes redes de comunicaciones ha aumentado con la incorporación de un gran número de dispositivos para mejorar la recolección de información y conseguir procesos más eficientes e inteligentes. Consecuentemente, el estudio de la integración de sistemas de seguridad en los propios dispositivos, entre ellos los microcontroladores, ha ido cobrando mayor importancia en los últimos años.

Este proyecto estudia la implementación del algoritmo AES (*Advanced Encryption Standard*) en la familia de microcontroladores PIC18F, específicamente en el PIC18F4520 utilizado en diversas prácticas tanto del Grado como del Máster en Ingeniería Industrial. Para ello, se realizan diversas simulaciones mediante el software MPLAB IDE y el compilador de C gratuito C18, ambas herramientas propiedad de Microchip Technology Inc.

Tras una breve introducción, se expone la versión inicial descargada de internet y las modificaciones realizadas para su posible compilación con las herramientas de las que se dispone. Más adelante, se describen los conceptos matemáticos básicos necesarios para el posterior entendimiento de la breve explicación que se realiza sobre el funcionamiento del algoritmo de cifrado AES. Por último, se desarrolla de forma detallada cada una de las modificaciones realizadas para la optimización del programa hasta llegar a la versión final.

Mediante la optimización se consigue disminuir considerablemente tanto el tiempo de ejecución como el uso de la memoria de programa (ROM). Con ello se demuestra la posibilidad de implementar el algoritmo de cifrado AES en esta familia de microcontroladores, permitiendo así una comunicación segura entre ellos.

Sumario

1. Glosario

AddRoundKey: función en la que se realiza una suma exclusiva entre la variable *Estado* y 16 valores determinados de la clave extendida.

AES: *Advanced Encryption Standard.* Algoritmo estándar de cifrado desde el año 2000, ganador del concurso internacional organizado por el NIST que tuvo lugar entre 1997 y 2000.

CBC: *Cipher Block Chaining Mode.* Modo de cifrado en el que el cifrado de cada bloque en que se divide el texto claro depende del contenido del anterior.

Cipher: conjunto de operaciones que transforman el texto claro en texto cifrado.

DES: *Data Encryption Standard.* Estándar de Encriptación de Datos que se utilizaba antes del algoritmo AES.

ECB: *Electronic Codebook Mode.* Modo de cifrado en el que el cifrado de cada bloque en que se divide el texto claro se realiza de forma separada, independientemente el uno del otro.

EEPROM: *Electrically Erasable Programmable Read-Only Memory.* Memoria ROM programable y con capacidad de ser borrada eléctricamente.

Bloque: agrupación de bits.

Estado: matriz de bytes con 4 filas y *Nb* columnas.

Flash: derivado de la memoria EEPROM que permite la lectura y escritura de múltiples posiciones de memoria en la misma operación.

GF: campo finito o campo de Galois, el cual contiene un número finito de elementos y en el que las operaciones de adición, producto, sustracción y división cumplen unas reglas determinadas.

Github: plataforma web de desarrollo colaborativo.

InvCipher: conjunto de operaciones que transforman el texto cifrado en texto claro.

InvMixColumns: transformación que actúa sobre cada columna de la variable *Estado* y que consiste en la multiplicación de éstas por un polinomio determinado, inverso al utilizado en *MixColumns*.

InvShiftRows: transformación que se aplica sobre cada fila de la variable *Estado* y que consiste en una rotación hacia la derecha un número determinado de bytes.

InvSubBytes: transformación que consiste en la substitución no lineal de cada byte de la variable *Estado* mediante la tabla *RS-Box*.

KeyExpansion: conjunto de operaciones mediante las que se genera la clave extendida a partir de la clave de cifrado.

MixColumns: transformación que actúa sobre cada columna de la variable *Estado* y que consiste en la multiplicación de éstas por un polinomio determinado.

Nb: número de columnas en la variable *Estado*.

NIST: *National Institute of Standards and Technology.* Instituto Nacional de Estándares y Tecnología que promueve la innovación y la competitividad de EE.UU. mediante el avance de la metrología, estándares y tecnología de manera que mejore la seguridad económica y la calidad de vida.

Nk: número de palabras (conjunto de 32 bits) en la clave.

Nr: número de rondas en el cifrado y descifrado.

PIC: *Peripheral Interface Controller*. Familia de microcontroladores fabricados por Microchip Technology Inc.

RAM: *Random Access Memory.* Memoria de acceso aleatorio en la que se suelen almacenar los datos.

Rcon: variable que interviene en la expansión de la clave y que depende de la posición de la clave extendida que se esté calculando.

RotWord: función utilizada en la expansión de clave que consiste en un rotación hacia la izquierda de los bytes dentro de una misma palabra.

RoundKey: variable en la que se guarda la clave extendida.

RS-Box: tabla mediante la que se realiza una substitución no lineal de un byte, inversa a la que se lleva a cabo mediante la tabla S-Box.

S-Box: tabla mediante la que se realiza una substitución no lineal de un byte.

ShiftRows: transformación que se aplica sobre cada fila de la variable *Estado* y que consiste en una rotación hacia la izquierda un número determinado de bytes.

Stdint.h: archivo de cabecera propio del estándar C99 en el que se definen varios tipos enteros.

SubBytes: transformación que consiste en la substitución no lineal de cada byte de la variable *Estado* mediante la tabla *S-Box*.

SubWord: función utilizada en la expansión de clave que substituye cada uno de los bytes de una palabra mediante la tabla *S-Box*.

XOR: operación suma exclusiva.

xtime: función que realiza la multiplicación de un byte por *x* módulo un polinomio irreducible de grado 8.

2. Introducción

2.1. Motivación

A lo largo de la historia, ha ido evolucionando la forma en la que se cifran los mensajes, desde la Escítala de los espartanos, pasando por la máquina enigma en la II Guerra Mundial, hasta los algoritmos de gran complejidad de la era informática [1].

La Real Academia Española define la criptografía como el “arte de escribir con clave secreta o de un modo enigmático” [2]. Mediante este arte se pretende proteger la información que contiene el mensaje de posibles interceptaciones, evitar el acceso de personas no autorizadas y la modificación del contenido, entre otros.

Con la irrupción de internet, la criptografía ha adquirido un papel importante en la sociedad actual. La banca on-line, protección de datos o comunicaciones inalámbricas serían sólo el principio de una larga lista de posibles aplicaciones. Últimamente es muy común escuchar noticias sobre hackers, ciberseguridad o cifrados en los mensajes de las diferentes redes sociales.

A nivel industrial, también se ha producido un gran desarrollo en las comunicaciones debido a la necesidad de respuestas en tiempo real de unidades productivas cada vez más automatizadas con sistemas de control cada vez más complejos. Consecuentemente, el análisis de seguridad informática en éstas se ha vuelto especialmente crítico, tanto a nivel de confidencialidad como de seguridad física. En los próximos años, se van a dedicar muchos esfuerzos para mejorar la ciberseguridad, dado el crecimiento exponencial de las comunicaciones causadas por el internet de las cosas.

Existen diferentes tipos de redes según el ámbito industrial [3]. Por ejemplo, en las redes de oficina, ventas, almacén o contabilidad, el tiempo de respuesta no es crítico pero el volumen de información que se transmite es muy elevado. Por el contrario, los buses de comunicación entre sensores/actuadores con los elementos de control respectivos tienen unas características muy diferentes. Deben ser de bajo coste, en tiempo real, sencillos, etc.

Para realizar la mayoría de estas comunicaciones, a medida que nos vamos acercando al mundo físico, adquieren más importancia los microcontroladores, circuitos integrados programables especializados en la interacción con el mundo real.

Son muy frecuentes los microcontroladores de 8 bits, ya que son fiables, baratos y capaces de realizar la mayoría de aplicaciones comunes, tanto industriales como de gran consumo. En la Figura 2.1 aparece uno de los muchos ejemplos posibles: el control de una caja fuerte. Como se puede observar, consta de un microcontrolador de 8 bits, un teclado de 16 teclas, un zumbador y una bobina.

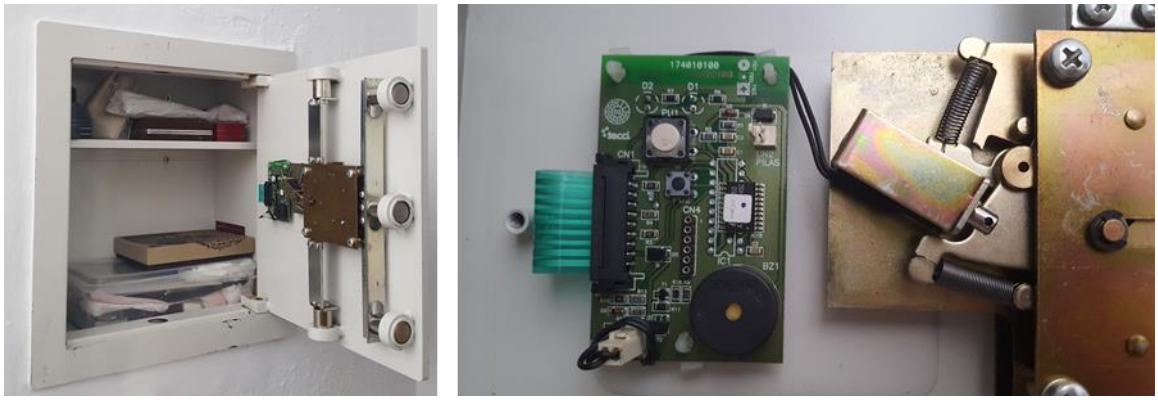


Figura 2.1: Ejemplo aplicación microcontrolador. Fuente: propia.

Dentro de las diferentes familias de microcontroladores, destacan los PIC, fabricados por Microchip Technology Inc., cuyo uso está muy extendido tanto a nivel académico como a nivel industrial.

2.2. Objetivos del proyecto

El objetivo del proyecto es implementar el algoritmo de cifrado AES en un microcontrolador de 8 bits de la familia PIC18F. Se modificará y se adecuará una versión obtenida de internet específicamente para esta familia de microcontroladores.

Dado que la capacidad de estos dispositivos es reducida en comparación con los que se utilizan en las computadoras, se optimizará el código de forma que se reduzca el tiempo de ejecución y el uso de la memoria, tanto de programa (ROM) como de datos (RAM).

Es común que estos dos objetivos entren en conflicto. Es decir, una reducción del tiempo resulta, la mayoría de veces, en un aumento de la memoria. Por este motivo, el proceso de optimización se dividirá en dos fases. La primera buscará ocupar el mínimo de memoria de programa posible y, la segunda, una disminución del tiempo de ejecución del programa.

2.3. Alcance del proyecto

Este proyecto no pretende realizar un estudio en profundidad del microcontrolador, sino ofrecer una implementación optimizada del algoritmo AES-128 que permita la implantación de seguridad de información en toda la familia de microcontroladores PIC18F.

Partiendo de una versión no optimizada del algoritmo AES-128, el proyecto concluye con la obtención de una versión optimizada de dicho algoritmo, ejecutándose en un microcontrolador de la familia PIC18F.

3. Antecedentes

Actualmente, existen multitud de webs en las que se puede encontrar documentación sobre casi cualquier tema. Se han buscado diferentes implementaciones por internet y, como punto de partida, se ha escogido la de un usuario de la plataforma de desarrollo colaborativo *Github* en la que se había colgado una implementación del AES-128 con un test de verificación incluido [4]. De entre los documentos descargados, se han utilizado los archivos *aes.c* y *aes.h*.

Para verificar el correcto funcionamiento de la versión mencionada se ha utilizado el software MPLAB IDE y el compilador de C gratuito C18, ambas herramientas propiedad de Microchip Technology Inc. En las simulaciones, tanto de la verificación como de las mejoras posteriores, era necesario especificar un microcontrolador y se ha escogido el PIC18F4520, dado que es el microcontrolador utilizado en las prácticas de diversas asignaturas del Grado y del Máster en Ingeniería Industrial. Además, se ha utilizado una frecuencia de procesador de 20 MHz.

3.1. Modificaciones

El principal problema a la hora de verificar su funcionamiento fue el compilador: la versión original de la página web utilizaba lenguaje C según el estándar C99, que no es 100% compatible con el lenguaje C del compilador C18. Por este motivo, se tuvieron que modificar algunos aspectos del programa.

Primero, se eliminó el archivo de cabecera *stdint.h,* propio del estándar C99. Para ello, se substituyeron todos los tipos de variable que éste incluía por sus equivalentes [5] [6]. En la Tabla 3.1, se muestran los tres reemplazados.

Stdint.h	Equivalencia
uint8_t	unsigned char
uint32_t	unsigned long int
uintptr_t	unsigned int

Tabla 3.1: Equivalencias tipos de variables stdint.h. Fuente: propia.

Además, se incluían dos modos de cifrado diferentes: ECB y CBC. Generalmente, los modos de cifrado se utilizan cuando el texto que se quiere cifrar excede la longitud máxima del bloque de cifrado e indican cómo aplicar repetidamente las operaciones de cifrado propias del algoritmo.

El *Electronic Codebook Mode* (ECB) consiste en cifrar los bloques por separado, independientemente el uno del otro (Figura 3.1). La principal ventaja es que permite cifrar los bloques paralelamente y acceder a ellos de forma aislada. Sin embargo, no es un método de seguridad elevada ya que siempre se obtiene el mismo resultado cuando se cifran bloques que contienen información idéntica.

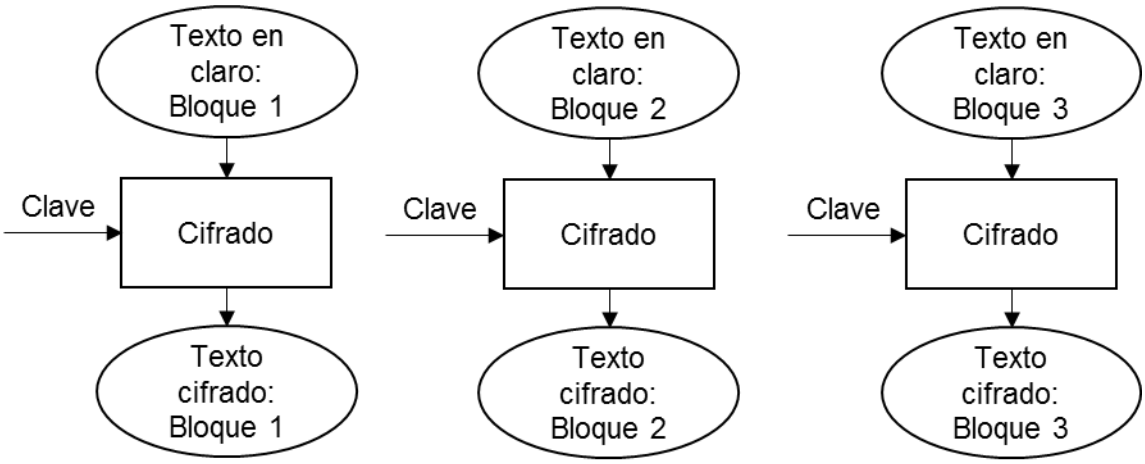


Figura 3.1: Cifrado en modo ECB. Fuente: propia.

En cambio, en el *Cipher Block Chaining Mode* (CBC), el primer paso es la operación suma exclusiva con el bloque de texto anterior ya cifrado. Así, cada bloque de texto depende del contenido del anterior. Para más inri, en el bloque inicial se utiliza un vector de inicialización que puede ser distinto en cada caso. En la Figura 3.2, se ilustra la forma de cifrado y descifrado en este modo.

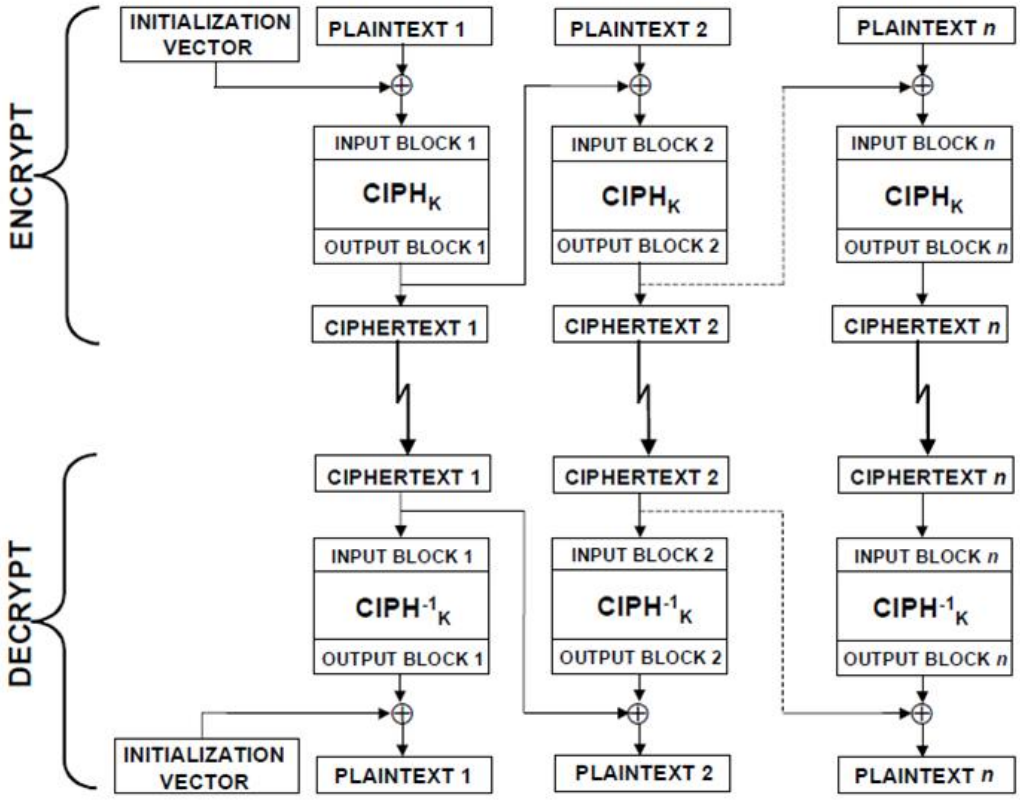


Figura 3.2: Modo CBC. Fuente: -[7].

Existen otros modos de cifrado además de los dos mencionados. Para profundizar sobre el tema, se recomienda la lectura de [7]. En este documento, se ha optado por el método básico ECB a partir del cual se pueden implementar otros modos mediante pequeñas modificaciones.

Así pues, se eliminó del programa todo lo referente al modo CBC, para centrarse únicamente en la implementación del modo ECB.

Por último, las tablas de 256 y 255 bytes que se definen al principio del programa, correspondientes a *S-Box*, *RS-Box* y *Rcon* (véase el capítulo 5), se tuvieron que definir como *static const rom* en vez de *static const*. Esto es debido a la sintaxis del compilador C18, que requiere de la directiva *static const rom* para poder declarar variables en memoria de programa.

En el Anexo 1, se muestra la versión inicial con todas las pequeñas modificaciones especificadas, de la cual se partirá para realizar las optimizaciones.

3.2. Criterios de optimización

Se valorarán las optimizaciones del programa mediante tres variables:

1. Memoria de programa (ROM)
2. Memoria de datos (RAM)
3. Tiempo de ejecución

Estos tres valores los muestra de forma automática en cada simulación el software MPLAB IDE de Microchip, mediante las opciones *Stopwatch* (donde aparece el tiempo, *time* en inglés, y las instrucciones de ciclo) y *Memory Usage Gauge* (donde aparece la Memoria de Programa y la Memoria RAM, *Program Memory* y *Data Memory* en inglés respectivamente).

Tal y como se ha mencionado anteriormente, el microcontrolador con el que se realizarán las simulaciones es el PIC18F4520 cuyas características de su memoria son:

- Memoria de Programa tipo Flash con capacidad de 32 kBytes
- Memoria RAM con capacidad de 1536 bytes
- Data EEPROM con capacidad de 256 bytes

En total, puede llegar a almacenar hasta 16384 instrucciones de una sola palabra que es el número que aparece en las simulaciones. Para consultar otros datos técnicos del microcontrolador se recomienda la lectura de [8].

En la Figura 3.3, se muestra el entorno de trabajo donde se pueden observar las dos ventanas (abajo a la derecha) donde aparecen las variables que se utilizarán como criterios de mejora. Específicamente, se muestra la versión inicial compilada.

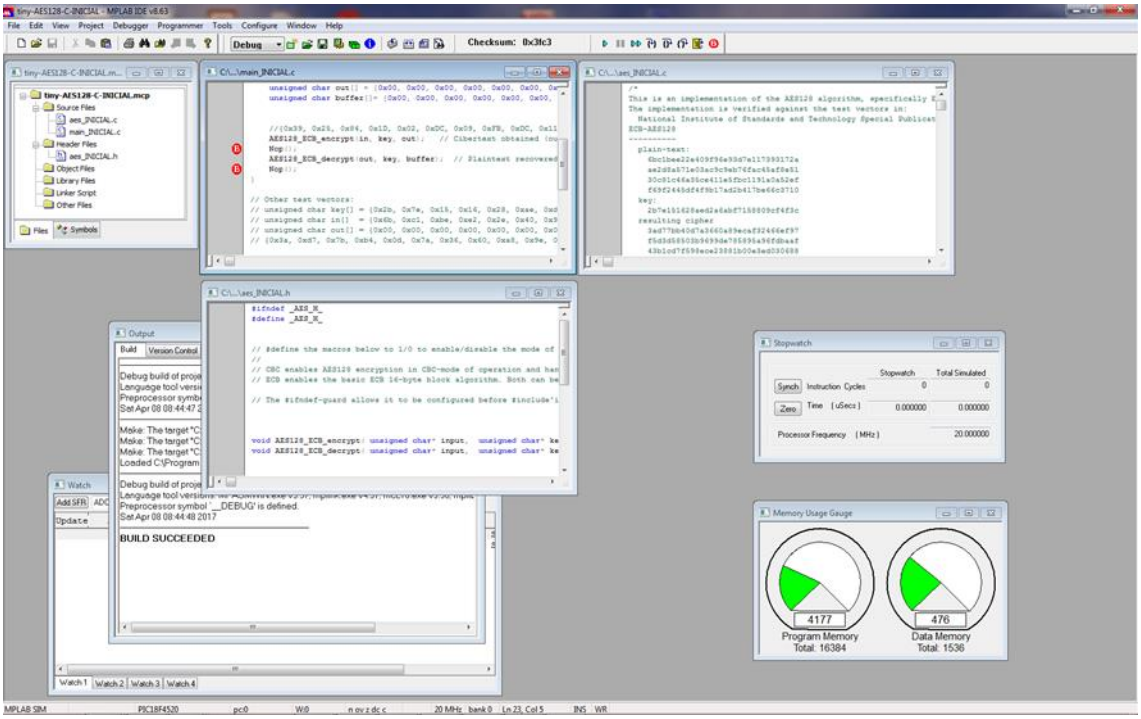


Figura 3.3: Entorno de trabajo con la versión de partida del AES-128. Fuente: propia.

3.3. Datos de partida

La Tabla 3.2 muestra los valores de los criterios correspondientes a la versión de partida, considerando un reloj de 20 MHz. Se ha dividido en tres columnas. La primera corresponde al cifrado y descifrado conjuntamente, que serán los valores con los que se comparará posteriormente.

	Inicial	Cifrar	Descifrar
Tiempo (ms)	54,0452	18,1270	35,9578
Memoria de Programa (instrucciones)	4177	2522	3332
Memoria RAM (bytes)	476	472	476

Tabla 3.2: Valores de partida. Fuente: propia.

Las dos columnas restantes simulan el caso en que el cifrado y descifrado se realizan en

dispositivos diferentes y, por lo tanto, en cada una de ellas únicamente aparece el código necesario para la aplicación correspondiente.

La suma de los dos tiempos se aproxima al total de la versión conjunta. Como se puede observar, es considerablemente más lento el descifrado que el cifrado. Esto es debido principalmente a la función *InvMixColumns*, ya que la implementación de la multiplicación de matrices que se realiza es con números mayores a los de *MixColumns*. En consecuencia, se debe llamar a la función *xtime* repetidamente, lo que produce un apreciable aumento del tiempo (véase el capítulo 5).

El mismo hecho influye en un incremento en el uso de memoria de programa. Éste también es causado por la necesidad de incluir tanto la tabla *RS-Box* como la *S-Box*, ya que esta última se utiliza en la rutina de expansión de clave. En cambio, en el cifrado únicamente se necesita la tabla *S-Box*.

4. Preliminares matemáticos

En este apartado se introducen los aspectos matemáticos principales necesarios para entender la implementación del algoritmo y las diferentes operaciones que se llevarán a cabo. Durante la explicación, los bytes serán interpretados como elementos del campo finito GF(2^8) y se definirán las operaciones suma y multiplicación específicas de dicho campo.

Como el objetivo es programar una versión optimizada del algoritmo, específica para su implementación en microcontroladores de la familia PIC18F, se estudiará la correspondencia de las operaciones en el campo finito GF(2^8) con las operaciones en binario.

Los elementos de un campo finito se pueden representar de distintas formas. Con el fin de conseguir el mejor entendimiento, se utilizará la representación polinómica acompañada en la mayoría de ejemplos de la correspondiente representación en binario y en hexadecimal. Así, un byte $b(x)$ (elemento del campo finito que se emplea en el algoritmo) se representa en forma polinómica, tal y como se muestra en la Ecuación 4.1.

$$b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 \qquad 4.1$$

Donde los coeficientes b_i con $0 \leq i < 8$ representan bits.

Para distinguir los números en decimal de la representación hexadecimal, se expresarán los bytes en hexadecimal entre corchetes. En la Tabla 4.1 se expone un ejemplo de las diferentes equivalencias.

Polinomio	Hexadecimal	Binario
$x^6 + x^4 + x^3 + x^1 + 1$	$\{5b\}$	0101 1011

Tabla 4.1: Equivalencias diferentes representaciones. Fuente: propia.

4.1. Adición

La adición de dos bytes en el campo finito GF(2^8) se define como la suma módulo 2 de los coeficientes correspondientes. Esta definición concuerda con la suma exclusiva (XOR) bit a

bit. Las propiedades de esta operación en binario aparecen en la Tabla 4.2.

⊕	**0**	**1**
0	0	1
1	1	0

Tabla 4.2: Propiedades suma exclusiva. Fuente: propia.

Una de las propiedades de esta operación es que el elemento inverso es él mismo. De aquí se deduce que la resta/sustracción es equivalente a la suma/adición. Esta característica ha sido destacada entre las demás porque, más adelante, será de utilidad. En la Tabla 4.3 se muestra un ejemplo de suma en las diferentes representaciones.

Representación	**Operación**
Polinómica	$(x^6 + x^4 + x^3 + x^1 + 1) + (x^6 + x^5 + x^3 + x^2 + x^1) = x^5 + x^4 + x^2 + 1$
Hexadecimal	$\{5b\} + \{6e\} = \{35\}$
Binario	$(0101\ 1011) + (0110\ 1110) = 0011\ 0101$

Tabla 4.3: Ejemplo adición en diferentes representaciones. Fuente: propia.

4.2. Multiplicación

La multiplicación en el campo finito GF(2^8) se define como el producto de un polinomio binario módulo otro irreducible de grado 8 ($m(x)$), es decir, aquél que sólo es divisible por él mismo y por 1. En la Tabla 4.4 se especifica dicho polinomio en el caso del algoritmo AES en las diferentes representaciones.

Representación	$\boldsymbol{m(x)}$
Polinomio	$x^8 + x^4 + x^3 + x + 1$
Hexadecimal	$\{11b\}$
Binario	$0001\ 0001\ 1011$

Tabla 4.4: Polinomio m(x) en las diferentes representaciones. Fuente: propia.

En el Ejemplo 1, tal y como se muestra en la siguiente página, se efectúa detalladamente la

operación de la Ecuación 4.2. Se ha dividido la operación en dos pasos: primero se calcula la multiplicación y, después, se reduce módulo $m(x)$.

$$\{57\} \cdot \{83\} = \{2b79\}\, mod\, m(x) = \{c1\} \qquad 4.2$$

Ejemplo 1 (Fuente: propia. Operaciones y resultados extraídos de [9]):

Multiplicación

$\cdot$ 01010111 10000011	$x^6 + x^4 + x^2 + x^1 + 1$ $\cdot$ $x^7 + x^1 + 1$
01010111	$x^6 + x^4 + x^2 + x^1 + 1$
01010111	$x^7 + x^5 + x^3 + x^2 + x^1$
101011100000	$x^{13} + x^{11} + x^9 + x^8 + x^7$
10101101111001	$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$

Módulo $m(x)$

10101101111001 $\oplus$ 10001101100000	$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$ $\oplus$ $(x^8 + x^4 + x^3 + x + 1) \cdot x^5$
00100000011001 $\oplus$ 100011011000	$x^{11} + x^4 + x^3 + 1$ $\oplus$ $(x^8 + x^4 + x^3 + x + 1) \cdot x^3$
000011000001	$x^7 + x^6 + 1$

Dado que la multiplicación así definida no tiene una correspondencia directa con las operaciones en binario, se define una función con el fin de facilitar la implementación del producto más adelante. Ésta se denomina $xtime()$ y es equivalente a la multiplicación de un elemento cualquiera del campo finito y x ($\{02\}$ en hexadecimal). En la Ecuación 4.3 se observa la definición de la función, siendo $b(x)$ la expresión definida en la Ecuación 4.1.

$$xtime\{b(x)\} = \{b(x)\} \cdot \{x\} = b_7x^8 + b_6x^7 + b_5x^6 + b_4x^5 + b_3x^4 + b_2x^3 + b_1x^2 + x \qquad 4.3$$

Si $b_7 = 0$, el resultado de la operación se mantiene, ya que es un polinomio de grado menor a 8. En cambio, si $b_7 = 1$, se debe substraer $m(x)$.

En binario, el producto de un número por x es equivalente a la rotación hacia la izquierda de dicho número. Por lo tanto, el producto en el campo finito GF(2^8) se puede definir de la

siguiente manera. Cuando $b_7 = 0$, el resultado será una simple rotación hacia la izquierda. En cambio, si $b_7 = 1$, la operación consistirá en una rotación a la izquierda seguida de una suma exclusiva bit a bit con $m(x)$.

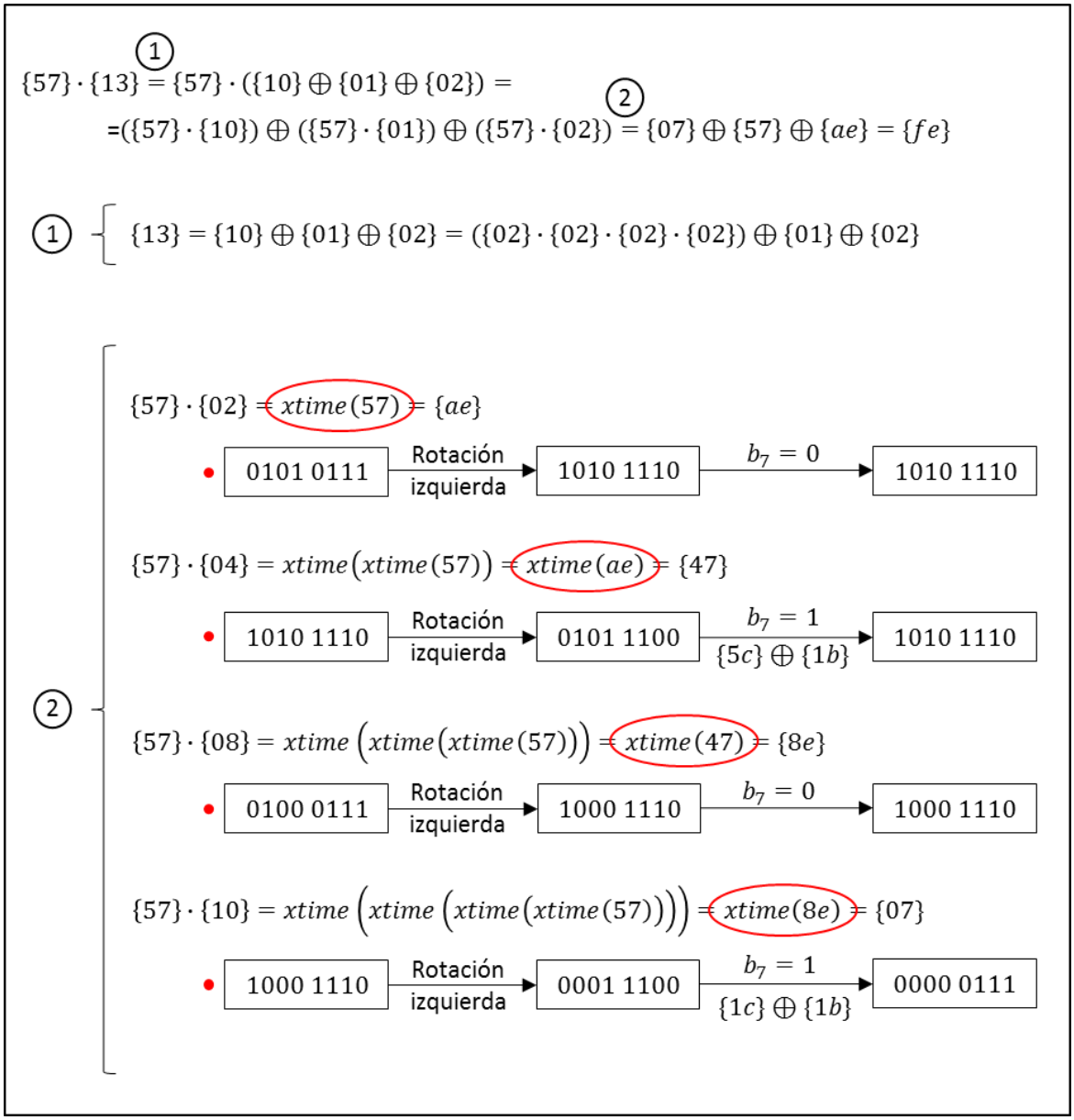
Todas las variables del programa se definirán como *unsigned char*, es decir, su tamaño será de 1 byte, con lo que el rango de valores será de 0 a 255. Este hecho es remarcable por dos motivos: el polinomio $m(x)$ no se puede representar con una única variable y el valor de b_7 se pierde cuando se realiza la rotación hacia la izquierda.

Por otro lado, no importa perder el valor de $b_7 = 1$ porque, en este caso, se tiene que substraer el polinomio $m(x)$ cuyo bit de mayor peso también es 1 y, por lo tanto, el resultado de la resta siempre es 0. En resumen, en la implementación de la función $xtime()$ en el caso de $b_7 = 1$, se rotará de igual forma hacia la izquierda pero se reducirá mediante el polinomio $\{1\mathrm{b}\}$ que es el resultado de $m(x) - \{100\}$.

En el Ejemplo 2 se explica detalladamente el desarrollo del producto expresado en la Ecuación 4.4.

$$\{57\} \cdot \{13\} = \{fe\} \qquad 4.4$$

Ejemplo 2 (Fuente: propia. Operaciones y resultados extraídos de [9]):



4.3. Polinomios

El algoritmo AES trabaja principalmente con palabras y no con bytes. Éstas pueden representarse como polinomios en el campo finito GF(2^8) de grado menor a 4, es decir, polinomios cuyos coeficientes son elementos del campo finito de trabajo (bytes). En la Ecuación 4.5 se representa un polinomio cualquiera $b(x)$.

$$b(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \qquad 4.5$$

La operación suma entre dos polinomios se define como la adición de los bytes

correspondientes a cada potencia de x. Esto es equivalente a la suma exclusiva de los bytes correspondientes en cada palabra.

Por otro lado, en la operación producto, es necesario definir otro polinomio de reducción, ya que los coeficientes son bytes y no bits como anteriormente. En el algoritmo AES se utiliza el polinomio $l(x)$, el cual se especifica en la Ecuación 4.6.

$$l(x) = x^4 + 1 \qquad 4.6$$

En el Ejemplo 3 se calcula el producto entre dos polinomios, $b(x)$ y $c(x)$, cuyo resultado es el polinomio $d(x)$, con coeficientes b_i, c_i y d_i respectivamente ($0 \leq i < 4$).

Ejemplo 3 (Fuente: propia. Operaciones y resultados extraídos de [9]):

$$(d_0 + d_1x + d_2x^2 + d_3x^3) =$$
$$= (b_0 + b_1x + b_2x^2 + b_3x^3) \times (c_0 + c_1x + c_2x^2 + c_3x^3) \; mod \; (x^4 + 1)$$

donde (1)(2):

$$d_0 = b_0 \cdot c_0 \oplus b_3 \cdot c_1 \oplus b_2 \cdot c_2 \oplus b_1 \cdot c_3$$
$$d_1 = b_0 \cdot c_1 \oplus b_1 \cdot c_0 \oplus b_3 \cdot c_2 \oplus b_2 \cdot c_3$$
$$d_2 = b_2 \cdot c_0 \oplus b_1 \cdot c_1 \oplus b_0 \cdot c_2 \oplus b_3 \cdot c_3$$
$$d_3 = b_3 \cdot c_0 \oplus b_2 \cdot c_1 \oplus b_1 \cdot c_2 \oplus b_0 \cdot c_3$$

(1) Multiplicación

$$(b_0 + b_1x + b_2x^2 + b_3x^3) \times (c_0 + c_1x + c_2x^2 + c_3x^3) =$$
$$= a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6$$

donde:

$$a_0 = b_0 \cdot c_0$$
$$a_1 = b_0 \cdot c_1 \oplus b_1 \cdot c_0$$
$$a_2 = b_2 \cdot c_0 \oplus b_1 \cdot c_1 \oplus b_0 \cdot c_2$$
$$a_3 = b_3 \cdot c_0 \oplus b_2 \cdot c_1 \oplus b_1 \cdot c_2 \oplus b_0 \cdot c_3$$
$$a_4 = b_3 \cdot c_1 \oplus b_2 \cdot c_2 \oplus b_1 \cdot c_3$$
$$a_5 = b_3 \cdot c_2 \oplus b_2 \cdot c_3$$
$$a_6 = b_3 \cdot c_3$$

(2) Módulo $l(x)$

$$(a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6) \; mod \; l(x)$$
$$= (a_0 \oplus a_4) + (a_1 \oplus a_5)x + (a_2 \oplus a_6)x^2 + (a_3)x^3$$

$$x^i \; mod \; (x^4 + 1) = x^{i \; mod \; 4}$$

El resultado obtenido en el Ejemplo 3 se puede representar en forma matricial (Ecuación 4.7).

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} b_0 & b_3 & b_2 & b_1 \\ b_1 & b_0 & b_3 & b_2 \\ b_2 & b_1 & b_0 & b_3 \\ b_3 & b_2 & b_1 & b_0 \end{bmatrix} \times \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} \quad 4.7$$

El polinomio $\mathrm{l}(\mathrm{x})$ no es irreducible en GF(2^8) y, por lo tanto, no todos los polinomios tienen un elemento inverso para la multiplicación módulo $\mathrm{l}(\mathrm{x})$. Este hecho es destacable porque a la hora de descifrar se procede de forma inversa al cifrado.

Para solucionar este inconveniente, en las funciones del algoritmo AES en las que se realiza esta operación, se multiplica por polinomios fijos de grado 4 que tienen inversa. Por ejemplo, en la función *MixColumns* del *Cipher* (cifrador) se multiplica por el polinomio fijo $\mathrm{b}(\mathrm{x}) = \{02\} + \{01\}\mathrm{x} + \{01\}\mathrm{x}^2 + \{03\}\mathrm{x}^3$, el inverso del cual es $\mathrm{b}^{-1}(\mathrm{x}) = \{0\mathrm{e}\} + \{09\}\mathrm{x} + \{0\mathrm{d}\}\mathrm{x}^2 + \{0\mathrm{b}\}\mathrm{x}^3$, que se utiliza en la función *InvMixColumns* del *InvCipher* (descifrador).

Para una explicación más detallada y formal de los aspectos matemáticos explicados en este capítulo, se recomienda la lectura del capítulo 2 de la referencia [10].

5. El algoritmo AES

Uno de los algoritmos de cifrado más utilizado y seguro actualmente es el AES. Según una noticia publicada por El País [11], el algoritmo AES es cuatro veces más vulnerable de lo que se creía desde su creación en el 2000. Aun así, la siguiente frase de la misma noticia resume claramente la seguridad del algoritmo: "un billón de ordenadores que pudieran cada uno probar mil millones de claves por segundo tardarían más de 2.000 millones de años en dar con una del sistema AES-128, y hay que tener en cuenta que las máquinas actuales sólo pueden probar 10 millones de claves por segundo".

Entre enero de 1997 y octubre del año 2000, el NIST (*National Institute of Standards and Technology*) organizó un concurso internacional con el objetivo de encontrar un sucesor para el DES (*Data Encryption Standard*) [10], el estándar de cifrado utilizado hasta el momento, debido a que se demostró que no era seguro. Se pretendía encontrar un nuevo algoritmo de cifrado potente, eficaz y fácil de implementar.

El ganador fue el algoritmo Rijndael, el nombre del cual es una combinación de sus dos autores belgas: Joan Daemen y Vincent Rijmen. Consiste en un sistema de cifrado simétrico por bloques en el que la longitud de la clave y la de los bloques puede variar independientemente entre cualquier múltiplo de 32 entre 128 y 256 bits. El adjetivo simétrico implica que se utiliza la misma clave para cifrar y descifrar.

La principal diferencia entre el algoritmo Rijndael y el estándar publicado por el gobierno estadounidense [9] es que la longitud de los bloques es fija e igual a 128 bits y la longitud de la clave puede variar únicamente entre 128, 192 y 256. El nuevo estándar adoptado se denomina AES (*Advanced Encryption Standard*) [10].

Tal y como se especifica en [12], los tres criterios principales que se tuvieron en cuenta a la hora de diseñar el algoritmo fueron:

- Resistencia contra todos los ataques conocidos.
- Velocidad y compactación del código en un amplio rango de plataformas.
- Simplicidad en el diseño.

Un esquema simplificado del funcionamiento del algoritmo AES se presenta en la Figura 5.1. Este capítulo se dedica a describir la estructura de éste y su funcionamiento. Para una

explicación más detallada, se recomienda la lectura de [10].

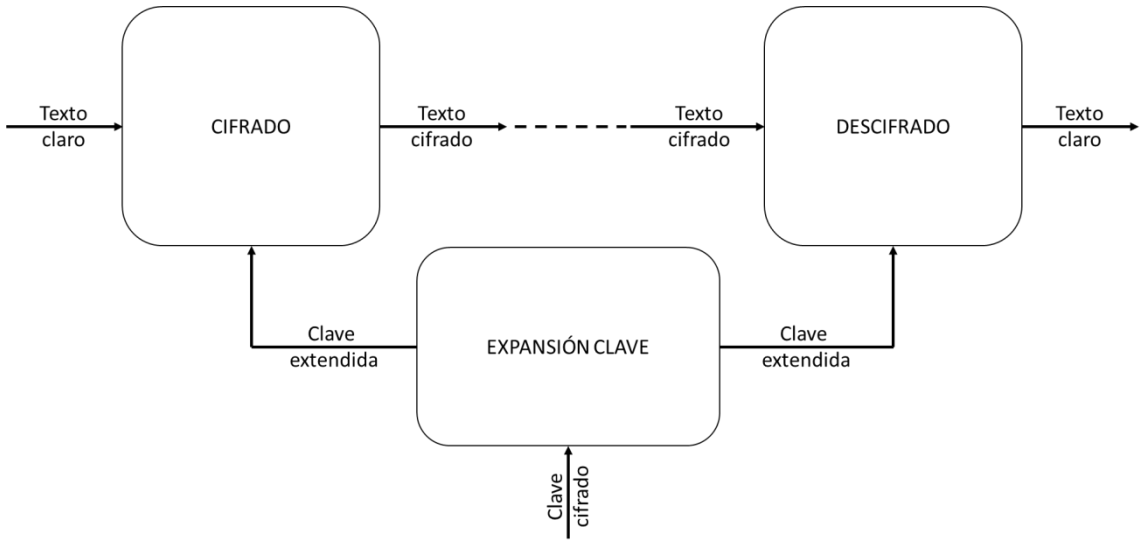


Figura 5.1: Esquema algoritmo AES. Fuente: propia.

5.1. Entradas y salidas

Tanto la *Entrada* como la *Salida* de la Figura 5.1 tienen un tamaño determinado de 128 bits. Es decir, son bloques que contienen un conjunto de 128 dígitos binarios (longitud del bloque) cuyos posibles valores son 0 ó 1. Si el texto que se pretende cifrar supera los 128 bits, se agrupará en bloques de esta longitud y se ejecutarán de forma sucesiva. En caso de que algún bloque quede incompleto, se completará con ceros o con algún valor predeterminado.

Por otro lado, la longitud de la clave de cifrado puede variar únicamente entre 128, 192 y 256 bits según el estándar. Así, se denomina AES-128, AES-192 y AES-256 a los algoritmos según la longitud de la clave que utilicen. Ésta se define mediante la variable *Nk,* que determina el número de palabras (conjunto de 32 bits, *word* en inglés) en la clave. Por lo tanto, los valores posibles de *Nk* son 4, 6 y 8, respectivamente.

El primer paso tanto en el cifrado como en el descifrado es copiar la *Entrada* en una variable auxiliar que se denominará *Estado* (*state* en inglés). En ella, se almacenarán los resultados intermedios de las rutinas de transformación, tanto en el cifrado como en el descifrado.

El algoritmo AES trabaja con palabras (conjuntos de 32 bits). Por este motivo, el *Estado* es una matriz 4x4, cuyo número de filas es fijo e igual a 4, representando así cada columna una

palabra. El número de columnas del *Estado* (*Nb*) en el estándar es constante y de valor igual a 4. Su cálculo se especifica en la Ecuación 5.1.

$$Nb = \frac{longitud\ del\ bloque}{32} = \frac{128}{32} = 4 \qquad 5.1$$

En la Figura 5.2, se especifica la manera en la que se copia la *Entrada* en el *Estado* y el *Estado* en la *Salida*.

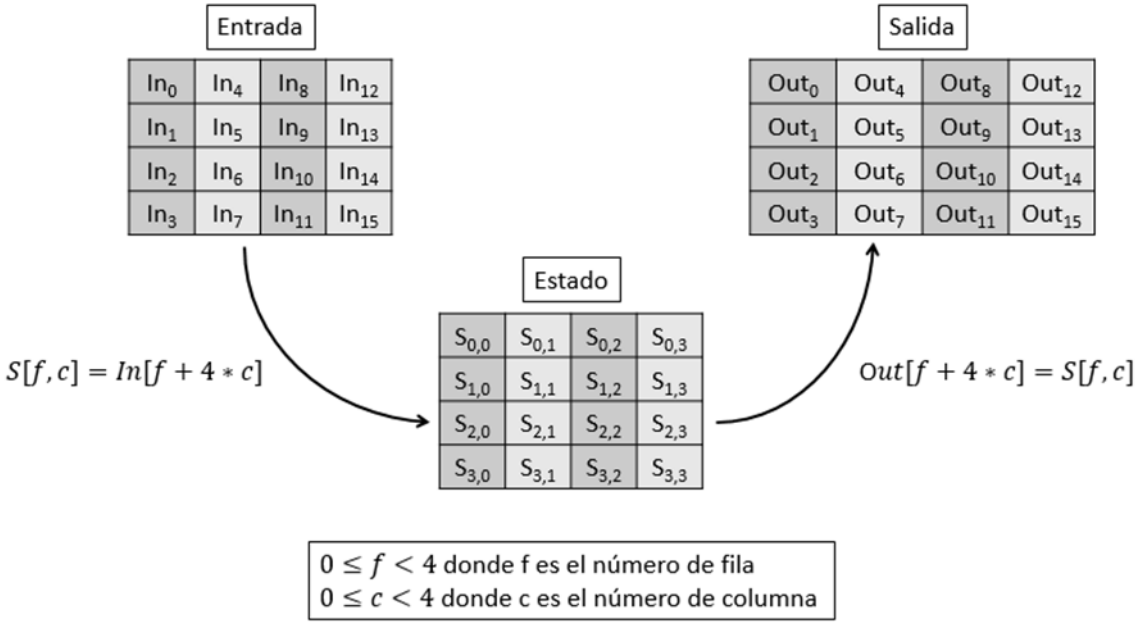


Figura 5.2: Copia de Entrada en Estado y Estado en Salida. Fuente: propia.

Según los valores de *Nb* (en el caso del estándar constante) y *Nk*, se define el número de rondas (*Nr*) que serán necesarias realizar en el cifrado y descifrado. Para que el cifrado sea más eficiente, en cada ronda se emplean diferentes valores generados a partir de la clave. Éstos se guardan en una variable (*RounKey*) y se generan a partir de una subrutina denominada *KeyExpansion*. La longitud de la clave extendida se define según la Ecuación 5.2.

$$Longitud\ clave\ extendida = (Nr + 1) * (tamaño\ bloque) \qquad 5.2$$

En la Tabla 5.1, se muestran los valores de *Nr* dependiendo de *Nb* y *Nk*, así como la longitud de la clave extendida en cada caso.

	Longitud clave (Nk)	Número columnas *Estado* (Nb)	Número rondas (Nr)	Longitud clave extendida
AES-128	4	4	10	176
AES-192	6	4	12	208
AES-256	8	4	14	240

Tabla 5.1: Valores Nr y longitud clave según Nb y Nk. Fuente: propia.

En este documento, se estudia únicamente la implementación del AES-128 en un PIC18F. Por lo tanto, en los siguientes apartados se utilizarán los valores predeterminados de éste, especificados en la Tabla 5.1.

5.2. Cifrado (*Cipher*)

El cifrado consiste en un bloque iterativo (*Cipher*) con un total de 10 rondas (*round*), aplicándose en cada ronda cuatro transformaciones diferentes sobre la variable *state*: *SubBytes*, *ShiftRows*, *MixColumns* y *AddRoundKey*. Sus entradas son un conjunto de 16 bytes correspondientes al texto sin cifrar y la clave extendida (176 bytes). La salida es el texto cifrado de igual tamaño que el de entrada.

En la Figura 5.3 se muestra un ejemplo de código, en el que la variable *w* representa la clave extendida. Como se puede observar, la ronda inicial y la final son diferentes del resto de rondas intermedias.

```
Cipher (byte entrada[16], byte salida[16], byte w[176])
{
        byte state[4,4];
        state=entrada;
        AddRoundKey (state, w[0,16]);
        for (round=1; round<10; round++){
                SubBytes (state);
                ShiftRows (state);
                MixColumns (state);
                AddRoundKey (state, w[round*4*4,(round+1)*4*4-1]);
        }
        SubBytes (state);
        ShiftRows (state);
        AddRoundKey (state, w[160,175);
        salida=state;
}
```

Figura 5.3: Código bloque de cifrado. Fuente: propia.

5.2.1. SubBytes

La transformación *SubBytes* consiste en una substitución no lineal de cada byte de *state* independientemente mediante una tabla que se denominará *S-Box* (Figura 5.4). Si, por ejemplo, se aplica la tabla *S-Box* al byte {3d}, el resultado sería {27}. Para una explicación detallada sobre la construcción de la tabla, se recomienda la lectura del capítulo 3 de [10].

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figura 5.4: Representación hexadecimal tabla S-Box (substitución byte xy). Fuente: -[9].

La Figura 5.5 ilustra el efecto de la función *SubBytes* en la variable *state*.

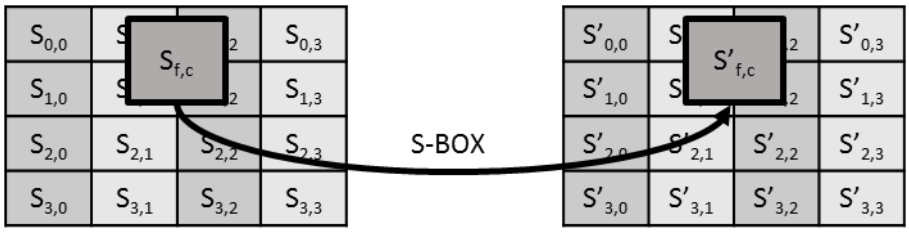


Figura 5.5: Aplicación SubBytes en state. Fuente: propia.

5.2.2. ShiftRows

La transformación *ShiftRows* se aplica sobre cada fila de la variable *state*. Consiste en una rotación hacia la izquierda de un número determinado de bytes, el cual depende de la posición de cada fila: la primera fila se mantiene igual, la segunda rota un byte hacia la izquierda, la tercera dos y la cuarta tres. La Figura 5.6 muestra el resultado de dicha transformación.

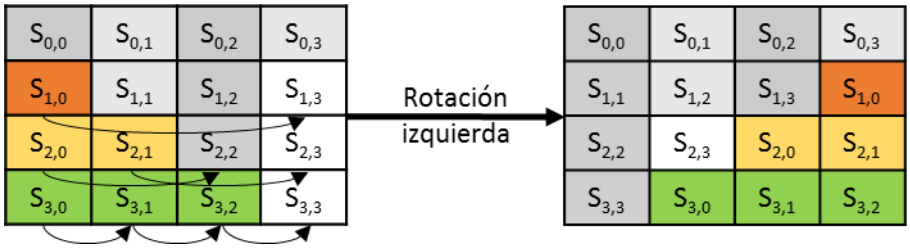
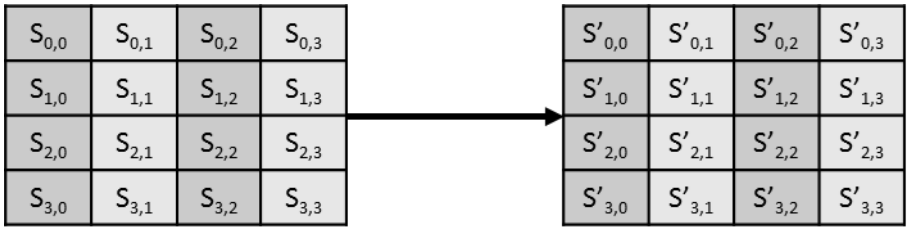


Figura 5.6: Aplicación ShiftRows en state. Fuente: propia.

5.2.3. MixColumnms

La transformación *MixColumns* actúa en cada columna de la variable *state,* considerándolas como un polinomio de grado 4 (véase la sección 4.3). Consiste en la multiplicación de cada una de ellas por el polinomio $\{02\} + \{01\}x + \{01\}x^2 + \{03\}x^3$. Tal y como se ha explicado en apartados anteriores, este producto se puede expresar en forma matricial. En la Figura 5.7, se representa el resulta de la transformación junto con la representación matricial de la multiplicación.



$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} para\ 0 \leq c < 4$$

Figura 5.7: Aplicación Mixcolumns en state. Fuente: propia.

En las Ecuaciones 5.3, 5.4, 5.5 y 5.6, se desarrolla el producto y se expresan cada uno de los bytes de una columna en función de $xtime()$.

$$s'_{0,c} = (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} =$$

$$= xtime(s_{0,c}) \oplus (xtime(s_{1,c}) \oplus s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \quad 5.3$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} =$$

$$= s_{0,c} \oplus xtime(s_{1,c}) \oplus (xtime(s_{2,c}) \oplus s_{2,c}) \oplus s_{3,c} \quad 5.4$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) =$$

$$= s_{0,c} \oplus s_{1,c} \oplus xtime(s_{2,c}) \oplus (xtime(s_{3,c}) \oplus s_{3,c}) \quad 5.5$$

$$s'_{3,c} = (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \cdot s_{3,c}) =$$

$$= (xtime(s_{0,c}) \oplus s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus xtime(s_{3,c}) \quad 5.6$$

5.2.4. AddRoundKey

En esta transformación es donde interviene la clave extendida. Consiste en la suma exclusiva bit a bit de la variable *state* y la clave extendida. En cada una de las rondas se

utiliza un conjunto de 16 bytes diferentes de la clave extendida (en la primera ronda se usan los 16 primeros, en la segundo los 16 siguientes y así sucesivamente).

La Figura 5.8 ilustra el funcionamiento de esta transformación.

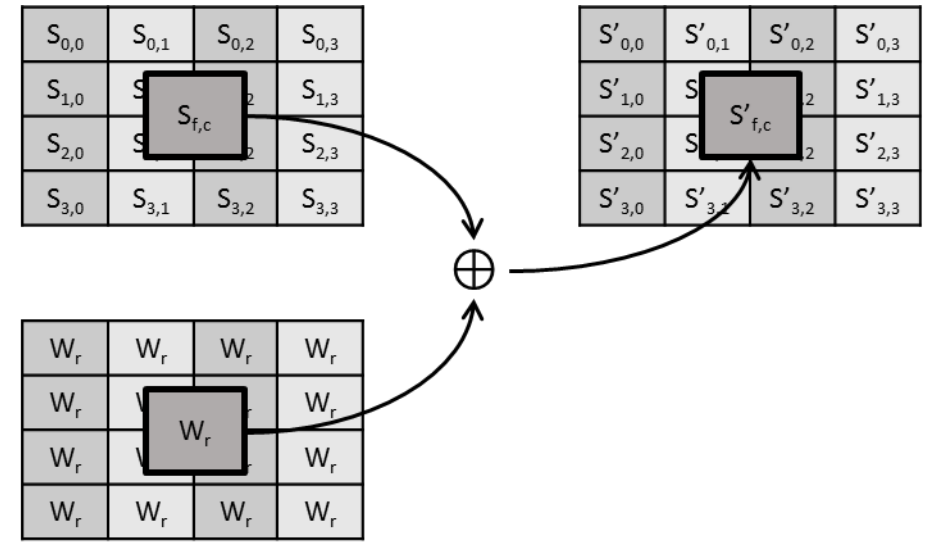


Figura 5.8: Aplicación AddRoundKey en state. Fuente: propia.

5.3. Descifrado (*InvCipher*)

Para llevar a cabo el descifrado, se siguen los mismos pasos que en el cifrado, pero en sentido inverso. Por lo tanto, también consiste en un bloque iterativo (*InvCipher*) con un total de 10 rondas, compuesta cada ronda por cuatro transformaciones inversas a las descritas en el apartado anterior: *InvShiftRows*, *InvSubBytes*, *InvMixColumns* y *AddRoundKey*.

Sus entradas son el texto cifrado de 16 bytes y la misma clave extendida que en el *Cipher* (176 bytes). La salida es el texto sin cifrar de la misma longitud que la entrada. En la Figura 5.9 se muestra un ejemplo del pseudocódigo.

```
InvCipher (byte entrada[16], byte salida[16], byte w[176])
{
        byte state[4,4];
        state=entrada;
        AddRoundKey (state, w[160,175]);
        for (round=9; round>0; round--){
                InvShiftRows (state);
                InvSubBytes (state);
                AddRoundKey (state, w[round*4*4,(round+1)*4*4-1]);
                InvMixColumns (state);
        }
        InvShiftRows (state);
        InvSubBytes (state);
        AddRoundKey (state, w[0,15);
        salida=state;
}
```

Figura 5.9: Código bloque de descifrado. Fuente: propia.

Como se puede observar, todas las transformaciones tienen su inversa correspondiente excepto *AddRoundKey*. Esto es debido a que se realiza la misma operación, pero en vez de utilizar los 16 primeros bytes en la primera ronda y los 16 siguientes en la que continúa, se empieza por los 16 últimos bytes.

5.3.1. InvShiftRows

De igual modo que en *ShiftRows* se efectúa una rotación hacia la izquierda en cada fila de la variable *state* dependiendo de la posición de ésta, en la transformación *InvShiftRows* se procede de forma inversa. Esto se traduce en una rotación hacia la derecha de cada fila el mismo número de bytes. En la Figura 5.10, se ilustra la transformación descrita.

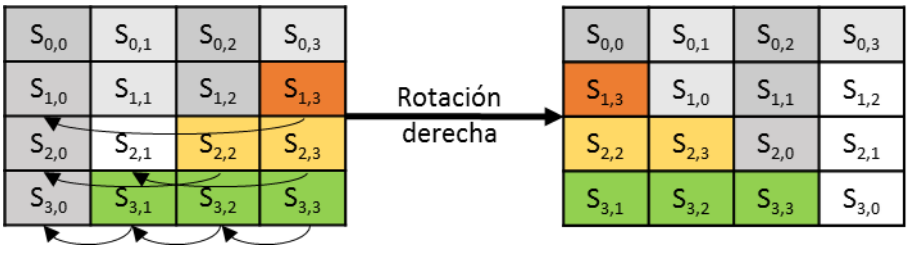


Figura 5.10: Aplicación InvShiftRows en state. Fuente: propia.

5.3.2. InvSubBytes

La transformación *InvSubBytes* es la inversa de *SubBytes*. Su funcionamiento es idéntico

pero realiza las substituciones independientes de los bytes de la variable *state* a través de la tabla inversa a *S-Box* que denominaremos *RS-Box* (Figura 5.11). De este modo, el byte {27} sería substituido por el {3d}.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figura 5.11: Representación hexadecimal tabla RS-Box (substitución byte xy). Fuente: -[9].

En la Figura 5.12, se ilustra el resultado de la transformación *InvSubBytes*.

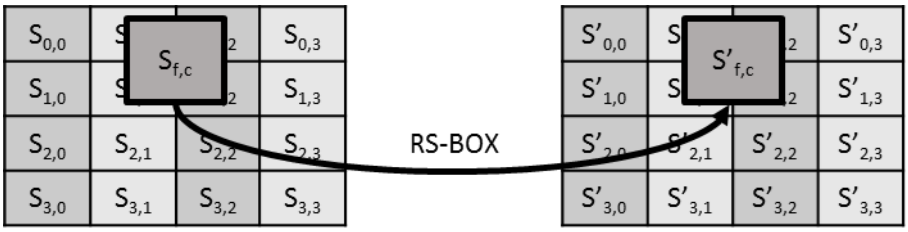
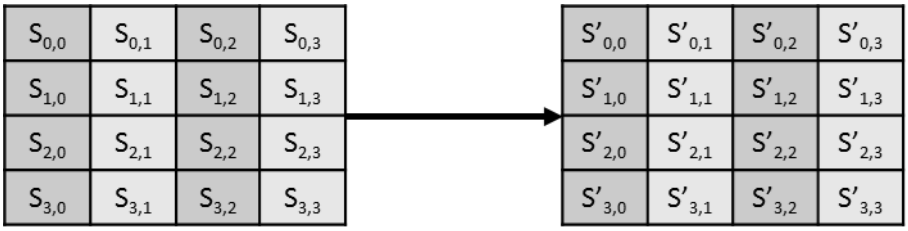


Figura 5.12: Aplicación InvSubBytes en state. Fuente: propia.

5.3.3. InvMixColumns

La transformación *InvMixColumns* consiste en el producto de cada una de las columnas de la variable *state,* consideradas como un polinomio de grado 4, por el polinomio inverso al utilizado en *MixColumns* ($\{0e\} + \{09\}x + \{0d\}x^2 + \{0b\}x^3$). En la Figura 5.13, se puede observar dicha transformación junto con la representación matricial del producto.



$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \cdot \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} para\ 0 \leq c < 4$$

Figura 5.13: Aplicación InvMixColumns en state. Fuente: propia.

Como se puede observar, en esta función interviene la multiplicación de un byte cualquiera $b(x)$ (Ecuación 4.1) por los cuatro valores que aparecen repetidamente en la matriz. Para facilitar la comprensión de la implementación en el programa que se analizará más adelante, en las Ecuaciones 5.7, 5.8, 5.9 y 5.10 se expresan estos cuatro productos a partir de la función $xtime()$.

$$\{0e\} \cdot b(x) = xtime\left(xtime\left(xtime(b(x))\right)\right) \oplus xtime\left(xtime(b(x))\right) \oplus xtime(b(x)) \quad 5.7$$

$$\{0b\} \cdot b(x) = xtime\left(xtime\left(xtime(b(x))\right)\right) \oplus xtime(b(x)) \oplus b(x) \quad 5.8$$

$$\{0d\} \cdot b(x) = xtime\left(xtime\left(xtime(b(x))\right)\right) \oplus xtime\left(xtime(b(x))\right) \oplus b(x) \quad 5.9$$

$$\{09\} \cdot b(x) = xtime\left(xtime\left(xtime(b(x))\right)\right) \oplus b(x) \quad 5.10$$

5.4. Expansión de la clave (*KeyExpansion*)

En el algoritmo AES-128, se genera una clave extendida de 176 bytes a partir de una clave dada de 16 bytes a través de la rutina denominada *KeyExpansion*. Ésta consta de dos funciones (*RotWord* y *SubWord*) que actúan sobre una variable auxiliar denominada $temp$, en la que se copia la última palabra de la clave extendida calculada y se aplican las diferentes operaciones.

En la Figura 5.14 se muestra un ejemplo del código correspondiente a *KeyExpansion*. Igual que en los ejemplos anteriores, la variable *w* representa la clave extendida. Cabe destacar que, en este caso, se ha expresado la clave extendida en palabras en vez de en bytes con el fin de facilitar la comprensión. Éste será un aspecto a tener en cuenta a la hora de implementar esta rutina, ya que se trabajará con bytes y no con palabras.

```
KeyExpansion (byte clave[16], word w[44])
{
        byte temp[4];
        i=0;
        while (i<4){
                w[i]=word (clave[4*i], clave[4*i+1], clave[4*i+2], clave[4*i+3]);
                i=i+1;
        }
        while (i<44){
                temp=w[i-1];
                if (i mod 4==0){
                        temp = SubWord(RotWord(temp))^Rcon[i/4];
                }
                w[i]=w[i-4]^temp;
                i=i+1;
        }
}
```

Figura 5.14: Código bloque expansión de clave. Fuente: propia.

Después de haber copiado la clave en las primeras 16 posiciones de la clave extendida, se aplican un seguido de operaciones y las funciones mencionadas.

5.4.1. RotWord

La función *RotWord* consiste en una rotación simple hacia la izquierda de los bytes guardados en *temp*. Esta transformación se ilustra en la Figura 5.15.

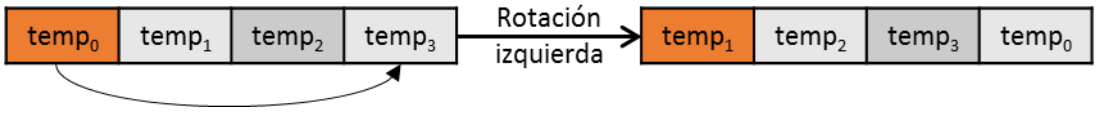


Figura 5.15: Aplicación RotWord en temp. Fuente: propia.

5.4.2. SubWord

En la función *SubWord* se substituye cada valor de la variable *temp* mediante la tabla *S-Box*.

En la Figura 5.16 está representada dicha transformación.

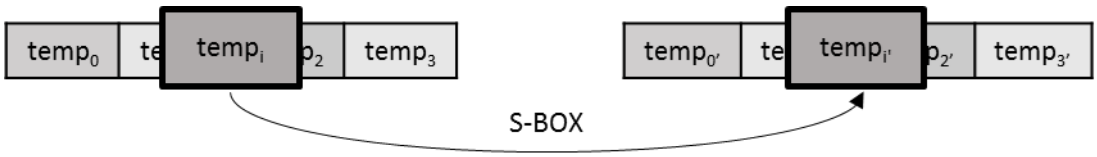


Figura 5.16: Aplicación SubWord en temp. Fuente: propia.

5.4.3. Rcon

La variable *Rcon* depende de la posición de la palabra de la clave extendida que se esté calculando (*i*). En la Ecuación 5.11 se muestra su expresión. El único byte de interés será el primero, ya que es el único diferente de 0 y, como se puede observar en la Figura 5.14, la variable se utiliza para llevar a cabo una suma exclusiva (ver propiedades suma exclusiva en la Tabla 4.2).

$$Rcon = \left[x^{i/4-1}, \{00\}, \{00\}, \{00\}\right] \text{ donde } 4 \leq i < 44 \qquad 5.11$$

Cabe mencionar, que *Rcon* solamente interviene cuando *i* es múltiplo de 4. En la Tabla 5.2 se han calcula los posibles valores de *Rcon* expresados en las diferentes representaciones.

$Rcon_{/i}$	Polinomio (mod $m(x)$)	Hexadecimal	Binaro
4	x^0	01	0000 0001
8	x^1	02	0000 0010
12	x^2	04	0000 0100
16	x^3	08	0000 1000
20	x^4	10	0001 0000
24	x^5	20	0010 0000
28	x^6	40	0100 0000
32	x^7	80	1000 0000
36	$x^4 + x^3 + x^1 + 1$	1B	0001 1011
40	$x^5 + x^4 + x^2 + x^1$	36	0011 0110

Tabla 5.2: Valores Rcon. Fuente: propia.

En la Figura 5.17 se muestra el diagrama de flujo del algoritmo AES.

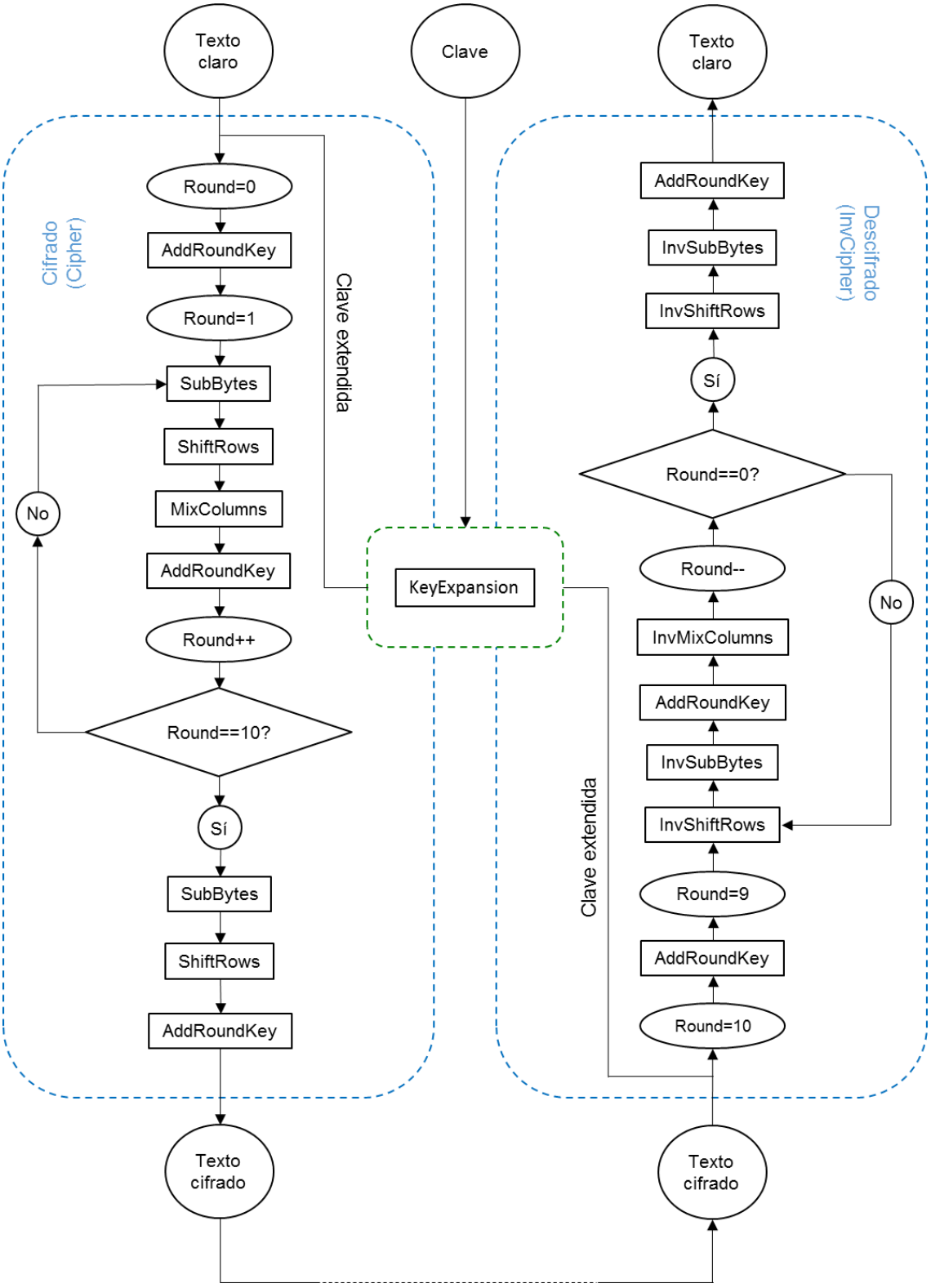


Figura 5.17: Esquema algoritmo AES. Fuente: propia.

6. Metodología

En este apartado se explicarán los diferentes intentos de mejora llevados a cabo hasta obtener una implementación óptima del algoritmo AES. Dado que los objetivos del proyecto son reducir tanto el tiempo como la Memoria de Programa (ROM) y la Memoria de Datos (RAM), puede haber diferentes versiones finales según el tipo de criterio que se priorice.

Al final de cada uno de los apartados de este capítulo se incluirá una tabla resumen de los valores alcanzados mediante la mejora explicada y de la última versión, para facilitar la comparación entre éstas. También se incluyen aquellos intentos que se han descartado y, en los casos en que sea posible, se adjunta una tabla con las mismas características para comprobar los datos por los que se ha excluido la alternativa.

Inicialmente se efectúan pequeñas modificaciones generales que disminuyen los valores de los tres criterios de mejora. Más adelante, siempre que se vea en conflicto, se opta por la opción que disminuye la Memoria de Programa. Finalmente, se modifica la estructura completa del programa para conseguir la versión con menor tiempo de ejecución posible.

6.1. Substitución bucles *for*

La primera modificación consiste en cambiar todos los bucles *for* por bucles *do…while*, los cuales tienen un tiempo de ejecución menor que los primeros. Ello se explica gracias a cómo el compilador traduce los bucles en instrucciones en ensamblador. A la hora de sustituir un tipo de bucle por otro, destacan dos factores: qué expresión booleana es necesaria y el lugar en el que se modifica el contador, ya que la condición se evalúa al inicio del bucle en el caso del *for* y al final en el *do…while*.

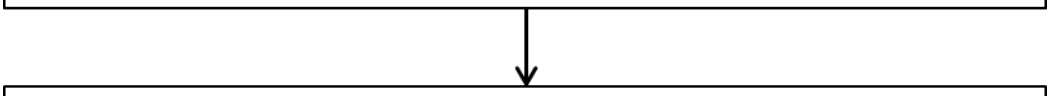
En este punto, también se han definido todas las variables como *unsigned char*, es decir, bytes cuyos valores pueden variar entre 0 y 255. Este hecho influye en los dos elementos mencionados en el párrafo anterior, tanto en la definición de la expresión booleana como en el lugar donde se modifica el contador, pues no es posible utilizar valores negativos.

Por último, cada uno de ellos se recorrerá de forma inversa, es decir, del máximo al mínimo. Ello se debe a la utilización de instrucciones en ensamblador del tipo “decrementa y salta”. En el caso del bucle que aparece en la función *Cipher* no se ha podido invertir el sentido en

el que se recorre porque cada ronda depende de la anterior.

Como ejemplo ilustrativo, la Figura 6.1 muestra cómo se ha efectuado el cambio en el bucle de la función *AddRoundKey*. Se puede observar que el valor de las variables auxiliares o contadores se ha asignado en el momento de definirlas, hecho que reduce también tanto el tiempo como la Memoria de Programa, ya que disminuye el número de instrucciones.

```
static void AddRoundKey(unsigned char round)
{
  unsigned char i,j;
  for(i=0;i<4;++i)
  {
    for(j = 0; j < 4; ++j)
    {
      (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];
    }
  }
}
```



```
static void AddRoundKey(unsigned char round)
{
  unsigned char i=4,j;
  do
  {
      i--;
      j=4;
    do
    {
      j--;
      (*state)[i][j] ^= RoundKey[round * Nb * 4 + i * Nb + j];
    }while (j>0);
  }while (i>0);
```

Figura 6.1: Ejemplo cambio bucle for por do...while. Fuente: propia.

En el caso concreto de la variable *j* de la Figura 6.1, no se ha asignado el valor 4 a la hora de definirla, pues se debía reiniciar cada vez antes de entrar en el segundo bucle. Este hecho muestra la necesidad de estudiar cada caso en particular.

En la Tabla 6.1, se puede observar como se ha reducido el tiempo de ejecución en 16,0452 ms, la Memoria de Programa en 633 instrucciones y la Memoria RAM en 3 bytes.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Inicial	54,0452	4177	476
Mejora 1	38,8000	3544	473

Tabla 6.1: Valores criterios en mejora 1. Fuente: propia.

6.2. Cambio expresiones *MixColumns*

Tal y como se explica anteriormente, la función *MixColumns* consiste en el producto de dos matrices. El principal problema a la hora de operar es que, a medida que se calcula cada una de las variables, se guardan los valores en la misma variable *state*, lo que obliga a guardar los valores iniciales en variables auxiliares. Para resolver este inconveniente, en la versión original se van guardando resultados intermedios en tres variables diferentes, *Tmp*, *Tm* y *t*, a partir de los cuales se calculan los valores finales.

En un primer momento se optó por utilizar las expresiones de las Ecuaciones 5.3, 5.4, 5.5 y 5.6 directamente. Para ello, se guardan los tres primeros valores de la columna correspondiente en cada ronda del bucle en las tres variables auxiliares de la versión original con las que, junto al último valor, se operaría para calcular los valores finales. Tal y como se puede observar en la Tabla 6.2, estas modificaciones reducen el tiempo en 0,2376 ms y la Memoria de Programa en 34 instrucciones.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 1	38,8000	3544	473
Mejora 2	38,5624	3510	473

Tabla 6.2: Valores criterios en mejora 2. Fuente: propia.

Más adelante, se valoró la alternativa de guardar los cuatro valores de la columna que se esté modificando en un vector de cuatro bytes con el que se realizarían las operaciones. El objetivo es reducir las veces que se debe acceder a la matriz *state* que, al tener doble indexación, el número de instrucciones para acceder a ella es mayor.

Mediante estos cambios, aumenta el tiempo en 0,1746 ms, pero se reduce la Memoria de Programa en 106 instrucciones, ambos valores respecto a la mejora 1. En la Tabla 6.3 se muestra los valores obtenidos.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 1	38,8000	3544	473
Mejora 3	38,9778	3438	473

Tabla 6.3: Valores criterios en mejora 3. Fuente: propia.

Aunque haya aumentado el tiempo, las siguientes mejoras se compararán con esta última versión expuesta ya que, de momento, la preferencia es la reducción de la Memoria de Programa. En la Figura 6.2, se pueden apreciar las diferencias entre el código inicial de la función *Mixcolumns* y las dos modificaciones explicadas.

```
static void MixColumns(void)
{
  unsigned char i;
  unsigned char Tmp,Tm,t;
  for(i = 0; i < 4; ++i)
  {
     t   = (*state)[i][0];
     Tmp = (*state)[i][0] ^ (*state)[i][1] ^ (*state)[i][2] ^ (*state)[i][3] ;
     Tm  = (*state)[i][0] ^ (*state)[i][1] ; Tm = xtime(Tm);
     (*state)[i][0] ^= Tm ^ Tmp ; Tm  = (*state)[i][1] ^ (*state)[i][2] ;
     Tm = xtime(Tm);  (*state)[i][1] ^= Tm ^ Tmp ;
     Tm  = (*state)[i][2] ^ (*state)[i][3] ; Tm = xtime(Tm);
     (*state)[i][2] ^= Tm ^ Tmp ;
     Tm  = (*state)[i][3] ^ t ; Tm = xtime(Tm); (*state)[i][3] ^= Tm ^ Tmp ;
  }
}
```

```
static void MixColumns(void)
{
  unsigned char i=4;
  unsigned char Tmp,Tm,t;
  do
  {
i--;
Tmp=(*state)[i][0];
Tm=(*state)[i][1];
t=(*state)[i][2];

(*state)[i][0]= xtime(Tmp^Tm)^Tm^t^(*state)[i][3];
(*state)[i][1]=xtime(t^Tm)^Tmp^t^(*state)[i][3];
(*state)[i][2]=xtime((*state)[i][3]^t)^(*state)[i][3]^Tmp^Tm;
(*state)[i][3]=xtime((*state)[i][3]^Tmp)^Tmp^Tm^t;
  }while (i>0);
}
```

```
static void MixColumns(void)
{
  unsigned char i=4,j;
  unsigned char t[4];
  do
  {
    i--;
    j=4;
    do
    {
      j--;
      t[j]=(*state)[i][j];
    }while(j>0);

    (*state)[i][0]=xtime(t[0]^t[1])^t[1]^t[2]^t[3];
    (*state)[i][1]=xtime(t[2]^t[1])^t[0]^t[2]^t[3];
    (*state)[i][2]=xtime(t[3]^t[2])^t[3]^t[0]^t[1];
    (*state)[i][3]=xtime(t[3]^t[0])^t[0]^t[1]^t[2];
  }while (i>0);
}
```

Figura 6.2: Fases transformación función MixColumns. Fuente: propia.

6.3. Construcción de bucles

En este apartado se busca reducir el número de instrucciones mediante la construcción de dos bucles con el formato *do…while,* al inicio y final de la expansión de clave (*KeyExpansion*) donde se copian cuatro valores en *RoundKey*. El tiempo de ejecución aumenta en 1,9244 ms, pero el uso de Memoria de Programa disminuye en 103 instrucciones. La Tabla 6.4 muestra los valores obtenidos.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 3	38,9778	3438	473
Mejora 4	40,8990	3335	473

Tabla 6.4: Valores criterios en mejora 4. Fuente: propia.

6.4. Función *SubWord*

Las instrucciones correspondientes a la función *SubWord* dentro de la expansión de la clave se repiten varias veces a lo largo de la subrutina. Por ese motivo, se ha construido una función aparte. Como la función utiliza la variable auxiliar *tempa*, ha sido necesario definirla como estática (*static)* para que fuese accesible en todas las funciones del programa y no solamente en la expansión de la clave.

Este hecho implica un aumento en la memoria RAM de 4 bytes (lo que ocupa *tempa*). Por el contrario, se consigue una disminución del tiempo de 1,0144 ms y en la Memoria de Programa de 84 instrucciones. Los datos correspondientes se encuentran en la Tabla 6.5.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 4	40,8990	3335	473
Mejora 5	39,8846	3251	477

Tabla 6.5: Valores criterios en mejora 5. Fuente: propia.

De igual modo se probó la posibilidad de escribir aparte la función *RotWord*. Sin embargo,

en este caso aumentaban los valores tanto del tiempo como de la Memoria de Programa y, por este motivo, se descartó la idea (Tabla 6.6).

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 5	39,8846	3251	477
Intento fallido 1	39,9406	3264	477

Tabla 6.6: Valores criterios en intento fallido 1. Fuente: propia.

6.5. Variable *i* como estática

Se puede observar como la variable auxiliar *i* se utiliza en la gran mayoría de las funciones. Este hecho es el que motiva la siguiente mejora: definir la variable *i* como estática (*static*). Con ello se consigue que el contenido de la variable se almacene únicamente una vez en la memoria.

En la Tabla 6.7, se muestra cómo esta modificación implica una reducción en los tres criterios: 4,6756 ms en tiempo de ejecución, 202 instrucciones en ocupación de Memoria de Programa y 17 bytes en Memoria RAM.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 5	39,8846	3251	477
Mejora 6	35,2650	3049	460

Tabla 6.7: Valores criterios en mejora 6. Fuente: propia.

6.6. Variable *j* como estática

Siguiendo el ejemplo del punto anterior, se define también la variable *j* como estática, ya que aparece en varias de las funciones que se utilizan. Además, se suprime la variable auxiliar *t* en *MixColumns* usando en su lugar la variable estática *tempa*. Esto es posible gracias a que *tempa* se utiliza únicamente en la expansión de clave, es decir, en el inicio del cifrado o

descifrado, y posteriormente no tiene ningún uso.

La modificación puede dificultar el entendimiento del programa al usar la misma variable en diferentes funciones. Sin embargo, la Memoria de Programa se reduce en 120 instrucciones, el tiempo en 0,7804 ms y la Memoria RAM se mantiene prácticamente igual (aumenta 1 byte). La Tabla 6.8 muestra los resultados obtenidos.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 6	35,2650	3049	460
Mejora 7	34,4846	2929	461

Tabla 6.8: Valores criterios en mejora 7. Fuente: propia.

6.7. Reducir tamaño *Rcon*

En la versión inicial, la variable *Rcon* se define como un conjunto de 255 bytes pero, tal y como se ha explicado en el apartado 5.4.3, los valores necesarios de *Rcon* para llevar a cabo la expansión de clave son 10 en el caso del algoritmo AES-128.

Por este motivo, se ha definido la variable *Rcon* como un conjunto de 11 bytes con un valor aleatorio en la primera posición. De este modo se evita tener que efectuar una operación suma cada vez que se utiliza la variable, ya que está incluida dentro del bucle de expansión de clave y, por lo tanto, depende del contador *i*. Si se añade un valor en la posición 0, cada número de ronda se corresponde con la posición del valor de *Rcon* necesario.

Por otro lado, se ha suprimido la variable auxiliar *k* en la expansión de clave, usando en su lugar la variable estática *j*. Del mismo modo que en el caso de *tempa*, este hecho dificulta el entendimiento del programa al utilizar una misma variable en diferentes lugares de la misma función. Al realizar este tipo de cambios, hay que tener claro cuándo se usa cada una y en qué momento está libre.

En la Tabla 6.9, se pueden observar los diferentes valores alcanzados. El tiempo de ejecución y la Memoria RAM se mantienen prácticamente igual y la Memoria de Programa se reduce considerablemente (131 instrucciones).

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 7	34,4846	2929	461
Mejora 8	34,4810	2798	461

Tabla 6.9: Valores criterios en mejora 8. Fuente: propia.

6.8. Definir Macros

Se han sustituido las dos funciones más simples por macros para así evitar las instrucciones de llamada. Éstas son *getSBoxValue* y *getSBoxInvert,* que únicamente cogen el valor correspondiente de la tabla S-Box o RS-Box respectivamente. En la Figura 6.3, se muestra el cambio realizado.

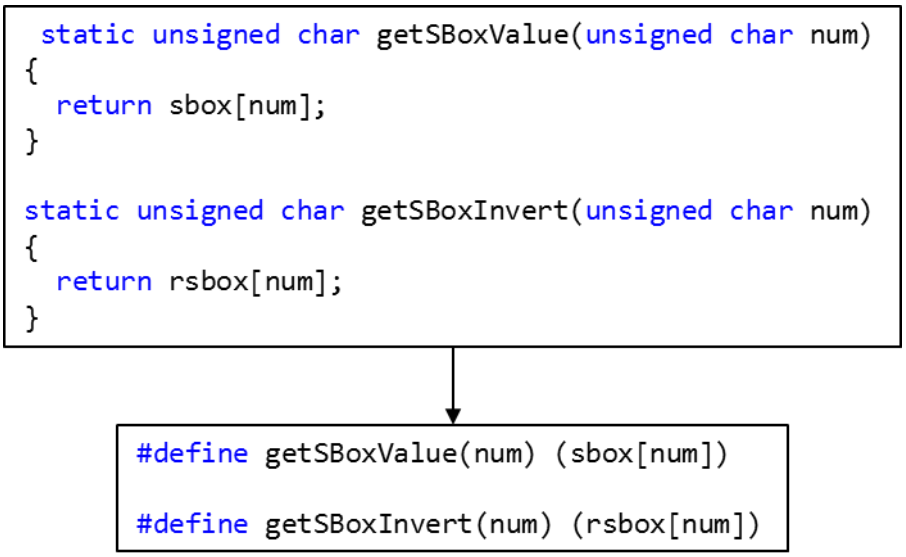


Figura 6.3: Cambio función por Macro. Fuente: propia.

Con esta modificación, se reduce 1,6614 ms el tiempo de ejecución, 76 bytes la Memoria de Programa y aumenta 1 byte la Memoria RAM. Los resultados obtenidos se resumen en la Tabla 6.10.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 8	34,4810	2798	461
Mejora 9	32,8196	2722	462

Tabla 6.10: Valores criterios en mejora 9. Fuente: propia.

6.9. Optimizador MPLAB

En el propio software MPLAB IDE, hay una opción que permite configurar la optimización que realiza el compilar cuando transforma el texto de lenguaje C a lenguaje ensamblador. Las posibilidades que ofrece se pueden observar en la pestaña central (*Build Options For Project*) de la Figura 6.4.

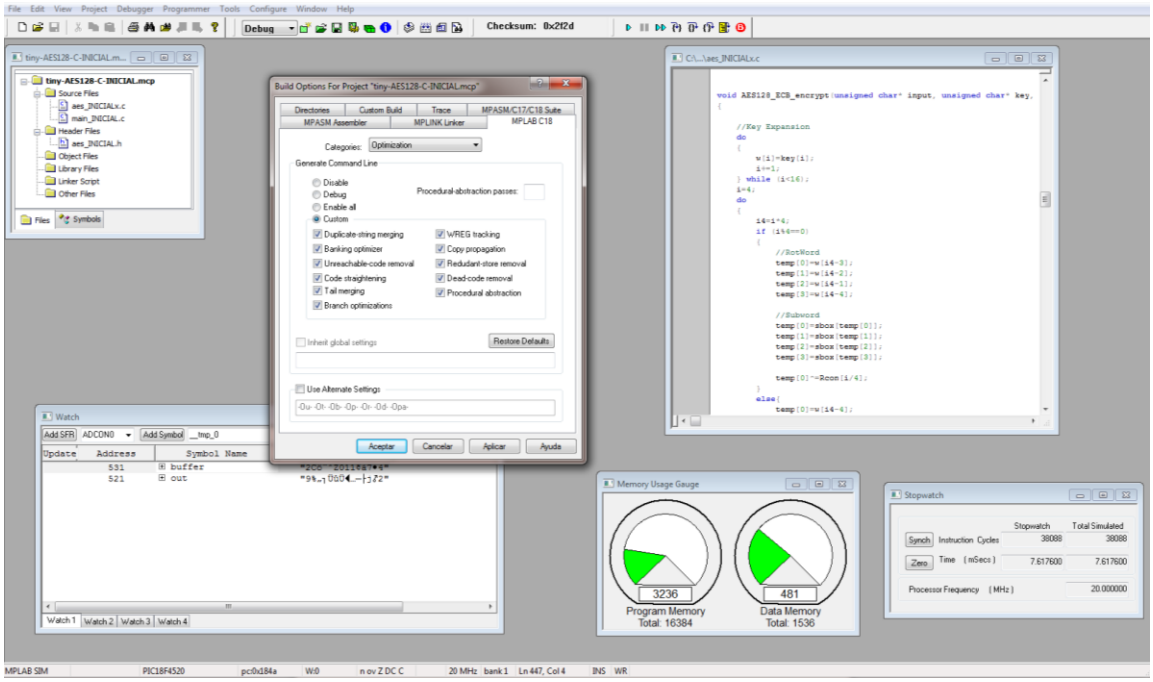


Figura 6.4: Posibilidad configurar optimizador C18 en MPLAB IDE. Fuente: propia.

El modo utilizado en todas las simulaciones realizadas hasta el momento es el denominado depurar (*debug* en inglés), pero en este apartado se intenta buscar la combinación de las diferentes opciones del modo *custom* con la que se consigan los menores valores de los criterios de optimización.

Sin embargo, tras varios intentos, los resultados obtenidos siempre han sido iguales o

peores de los que se partía. Cabe destacar que, si se mantenía únicamente la opción "optimizador de bancos" (*Banking optimizer* en inglés), los resultados se mantenían. Con cualquier otra combinación en la que se desactivase dicha opción, los valores alcanzados empeoraban.

6.10. Tabla *xtime*

Uno de los puntos críticos dentro del código (cuello de botella) es la función *xtime*, ya que es llamada un número considerable de veces, especialmente en *InvMixColumns*. Por este motivo, en este apartado se estudia detenidamente la implementación de esta función.

Tal y como se ha explicado en el apartado 4.2, la función *xtime* debe distinguir entre dos casos: $b_7 = 0$ o $b_7 = 1$. La implementación de la versión original se muestra en la Figura 6.5. Como se puede observar, esta distinción la hace rotando siete veces hacia la derecha el número de estudio. Si $b_7 = 0$, el resultado de la rotación es 0 y el de la suma exclusiva con $\{1b\}$ también, con lo que se mantiene la rotación simple hacia la izquierda. En cambio, si $b_7 = 1$, el resultado de la rotación es 1 y, consecuentemente, la rotación simple hacia la izquierda irá seguida de la suma exclusiva con $\{1b\}$.

```
static unsigned char xtime(unsigned char x)
{
  return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
}
```

Figura 6.5: Implementación función xtime en la versión original. Fuente: propia.

La primera opción contemplada es estudiar la forma de que no se tengan que realizar todas las operaciones cada vez que se ejecute la función mediante el uso de un condicional. En la Figura 6.6 se muestra la implementación planteada.

```
static unsigned char xtime(unsigned char x)
{
if (x & 0x80){
  return (x<<1 ^ 0x1b);
}
return x<<1;
}
```

Figura 6.6: Alternativa implementación función xtime. Fuente: propia.

Los valores obtenidos aparecen en la Tabla 6.11. Como se puede observar, la Memoria RAM se mantiene constante y la Memoria de Programa se reduce solamente 2 bytes, pero el tiempo de ejecución disminuye 2,736 ms.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 9	32,8196	2722	462
Mejora 10	30,0836	2720	462

Tabla 6.11: Valores criterios en mejora 10. Fuente: propia.

La segunda opción es construir una tabla con los resultados correspondientes a aplicar la función *xtime* a los 256 valores posibles que puede tener como entrada dicha función. En el Anexo 2.1 se muestran los valores obtenidos. La ventaja es que las instrucciones se reducen a coger un valor de la tabla disminuyendo así el tiempo de ejecución, pero el inconveniente es el espacio que ocupa ésta en la memoria.

Mediante esta modificación, se consigue reducir el tiempo en 3,7512 ms, pero la Memoria de Programa aumenta en 794 instrucciones y la Memoria RAM en 8 bytes. Los resultados se muestran en la Tabla 6.12.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 10	30,0836	2720	462
Mejora 11	26,3324	3514	470

Tabla 6.12: Valores criterios en mejora 11. Fuente: propia.

Dado el aumento considerable en el uso de memoria, el objetivo principal en los siguientes apartados será reducir al máximo el tiempo de ejecución, aunque suponga un aumento en la ocupación de la memoria. Por lo tanto, la mejora 10 es aquella con la que se ha conseguido el menor uso de memoria.

6.11. Nuevo planteamiento

Tal y como se ha mencionado en el apartado anterior, el objetivo principal en las siguientes modificaciones es reducir el tiempo de ejecución. Para ello, se ha vuelto a reescribir todo el

programa: se han eliminado las macros que substituían los valores de las tablas *S-Box*, *RS-Box* y *xtime*, accediendo a ellas directamente; la variable *state* se ha definido como un conjunto de 16 bytes en vez de una matriz para que el acceso a ésta sea más rápido; y se han evitado, en la medida de lo posible, los bucles y llamadas a funciones.

Tras varios intentos, los resultados obtenidos se muestran en la Tabla 6.13. Si se compara con la mejora 11, ya que se ha mantenido la tabla *xtime*, el tiempo de ejecución se reduce en 15,0572 ms y la Memoria de Programa en 338 instrucciones, pero la Memoria RAM aumenta en 15 bytes.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 11	26,3324	3514	470
Mejora 12	11,2752	3176	485

Tabla 6.13: valores criterios en mejora 12. Fuente: propia.

A través de los resultados obtenidos se comprueba que, en estos dispositivos de poca capacidad de cálculo, el rendimiento es mayor si se escribe el código de forma secuencial, en vez de estructurarlo mediante funciones y bucles.

6.12. Tablas multiplicación

Si se estudia el código en lenguaje ensamblador que proporciona el propio programa MPLAB IDE, se observa fácilmente que el principal cuello de botella se encuentra en la función *InvMixColumns*. Como ya se ha comentado anteriormente, esto es debido a que llama una gran cantidad de veces a la función *xtime*.

Por este motivo, se ha planteado la posibilidad de calcular 4 tablas de 256 bytes con los valores de la multiplicación de los 4 números que aparecen en la función *InvMixColumns*: $\{03\}$, $\{0b\}$, $\{0d\}$ y $\{09\}$. Éstas se llamarán *mul0e*, *mult0b*, *mult0d* y *mult09* respectivamente. En los Anexos 2.2, 2.3, 2.4 y 2.5 se encuentran los resultados obtenidos, respectivamente.

Mediante esta modificación, se ha conseguido reducir el tiempo en 3,6576 ms y la Memoria RAM en 4 bytes, pero la Memoria de Programa ha aumentado en 60 instrucciones. En la Tabla 6.14, se pueden observar los resultados obtenidos.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Mejora 12	11,2752	3176	485
Mejora 13	7,6176	3236	481

Tabla 6.14: Valores criterios en mejora 13. Fuente: propia.

Cabe destacar como, aun incluyendo 4 tablas de características similares a *xtime*, el aumento de Memoria de Programa es casi 6 veces menor en comparación con el que supone la modificación entre la mejora 10 y 11, en la que se añade solamente una tabla. Este hecho es debido a que, aunque se ocupe bastante espacio incluyendo las tablas, el número de instrucciones se reduce drásticamente: se pasa de llamar 72 veces a la función *xtime* junto con las operaciones matemáticas correspondientes, a coger únicamente 16 veces un valor de una tabla.

6.13. Tabla resumen

En la Tabla 6.15, se muestran las diferentes modificaciones realizadas. Se pueden observar tres mínimos diferentes: la mejora 6 en la Memoria RAM, la mejora 10 en la Memoria de Programa y la mejora 13 en el tiempo de ejecución.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Inicial	54,0452	4177	476
Mejora 1	38,8000	3544	473
Mejora 2	38,5624	3510	473
Mejora 3	38,9778	3438	473
Mejora 4	40,899	3335	476
Mejora 5	39,8846	3251	477
Mejora 6	35,2650	3049	460
Mejora 7	34,4846	2929	461
Mejora 8	34,4810	2798	461
Mejora 9	32,8196	2722	462
Mejora 10	30,0836	2720	462
Mejora 11	26,3324	3514	470
Mejora 12	11,2752	3176	485
Mejora 13	7,6176	3236	481

Tabla 6.15: Resumen mejoras. Fuente: propia.

7. Solución alcanzada

De entre los tres mínimos alcanzados (apartado 6.13), se destacan dos versiones: la mejora 10 con el mínimo uso de Memoria de Programa y la mejora 13 con el mínimo tiempo de ejecución. Se considerarán las dos soluciones finales de la optimización y se denominarán solución 1 y 2, respectivamente. Según la aplicación del microcontrolador y el objetivo que se busque, se puede considerar una opción u otra.

La mejora 6, correspondiente al mínimo uso de Memoria RAM, no se ha considerado como solución final, ya que la variación es bastante reducida. Entre la versión con el mínimo de Memoria RAM (460 bytes en la mejora 6) y con el máximo (485 bytes en la mejora 12), solamente hay una diferencia de 25 bytes, lo que supone una variación de menos del 2% respecto al total. Por este motivo, no se contempla entre las soluciones finales.

En la Tabla 7.1, se muestran los resultados de ambas junto con la inicial. Además, se especifican las reducciones en tanto por ciento respecto a los valores de la versión de partida. Cabe destacar que, mediante la solución 1, se consigue una reducción del 8,9% en el uso de la Memoria de Programa total y, con la solución 2, del 5,7%. Por otro lado, la diferencia en los tiempos de ejecución de ambas es considerable.

	Tiempo (ms)	Memoria de Programa (instrucciones)	Memoria RAM (bytes)
Inicial	54,0452	4177	476
Solución 1	30,0836	2720	462
Reducción	40,34%	34,88%	2,94%
Solución 2	7,6176	3236	481
Reducción	85,91%	22,53%	-1,05%

Tabla 7.1: Soluciones alcanzadas. Fuente: propia.

Dado que, en una aplicación real, se deberá emplear el algoritmo repetidamente, la disminución del tiempo tiene un impacto muy elevado en el rendimiento del proceso. Por este motivo, la solución óptima final sería la 2. En el Anexo 3, se adjunta el código correspondiente al archivo aes.c.

En la Tabla 7.2, se puede apreciar la optimización alcanzada en el cifrado y descifrado

respecto a la versión inicial de forma separada. La reducción en el tiempo supera el 80% y de la Memoria de Programa casi alcanza el 40 % en ambas aplicaciones. También se puede observar como el aumento de Memoria RAM no supera el 0,1% en ninguna de las dos.

	INICIAL		SOLUCIÓN 2	
	Encriptar	Desencriptar	Encriptar	Desencriptar
Tiempo (ms)	18,1270	35,9578	3,1888	4,5932
Memoria de Programa (instrucciones)	2522	3332	1572	2089
Memoria RAM (bytes)	472	476	479	489

Tabla 7.2: Comparación valores versión inicial y final. Fuente: propia.

8. Validación

Para comprobar el correcto funcionamiento de las dos versiones solución alcanzadas, se han utilizado los tests especificados en el AESAVS (*Advanced Encryption Standard Algorithm Validation Suite*) [13]. Tal y como se especifica en el mismo documento [13], el AESAVS utiliza una muestra estadística y, por lo tanto, no implica una comprobación del 100% con el estándar.

Se pueden distinguir dos tipos de tests: aquéllos en los que varía el texto claro, pero se usa la misma clave, y aquéllos en los que se varía la clave, pero se mantiene el mismo texto de entrada. Ambos están diseñados para ayudar en la detección de errores accidentales en la implementación y no para comprobar la seguridad.

Dado que, en el presente documento, se ha estudiado la implementación del AES-128 en modo ECB, se han considerado únicamente los tests correspondientes.

9. Presupuesto económico

En este proyecto se pueden distinguir principalmente dos gastos: las horas empleadas en su ejecución y el ordenador utilizado tanto en la investigación como en la programación y simulación.

Las horas de dedicación se pueden resumir en tres aspectos diferentes: familiarización con la programación en c, entendimiento del algoritmo y optimización del código. Si se estiman un total de 300 horas y que cada hora supone una remuneración de 40 €, el gasto en personal supondría 12.000 €. En la Tabla 9.1, se muestra el desglose de las horas correspondientes.

Actividad	Horas	€/h	Total
Lenguaje C	70	40	2.800 €
Estudio algoritmo AES	80	40	3.200 €
Optimización	150	40	6.000 €
			12.000 €

Tabla 9.1: Desglose presupuesto horas de trabajo. Fuente: propia.

En la totalidad de las horas dedicadas a la elaboración del proyecto, se emplea el ordenador. El utilizado es uno de características similares a las especificadas en [14] cuyo precio asciende a 1.000 € aproximadamente. Es remarcable que se podría haber utilizado un ordenador (portátil o de sobremesa) más económico, cualquiera en el que se pudiese instalar el programa utilizado en las simulaciones (MPLAB IDE).

Considerando que la vida media de un ordenador es de 5 años, en el supuesto caso de que se use 8 horas diarias, la inversión de esta partida sería de 21 €. Los datos necesarios para el cálculo aparecen en la Tabla 9.2.

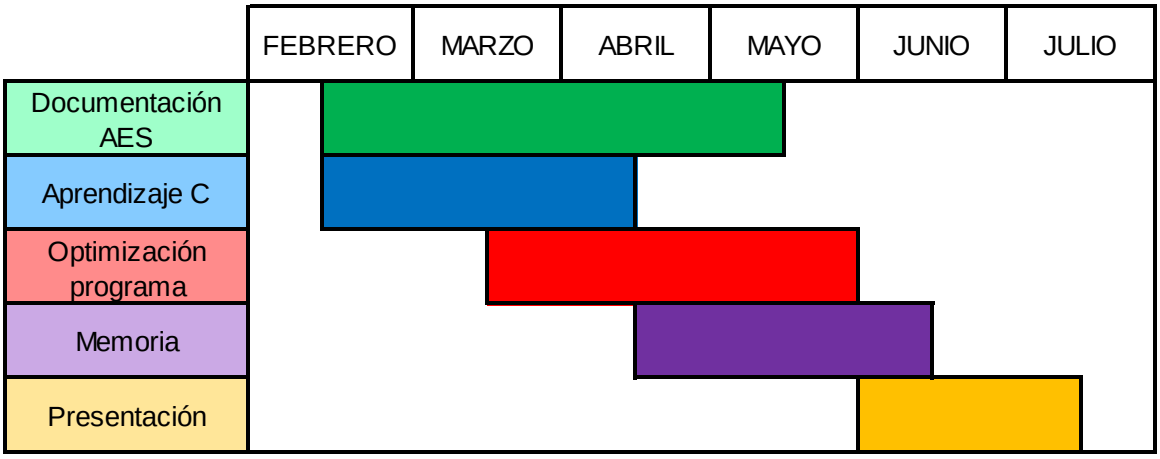
Herramienta	Precio	Duración		Horas	Inversión
Ordenador	1.000 €	5 años	14.600 h	300	**20,55 €**

Tabla 9.2: Cálculo amortización ordenador. Fuente: propia.

Si se suman las dos partidas, el presupuesto total del proyecto es de 12.021 € aproximadamente. Aun así, el impacto en el precio del producto sería muy reducido si se considera que el mismo código se puede utilizar en todos los productos que usen el mismo microcontrolador o alguno con características similares, ya que la adaptación es sencilla.

10. Planificación

La planificación del proyecto se divide en tres bloques principales: documentación sobre el algoritmo AES, aprendizaje de la programación en lenguaje C y la optimización del programa. Además, se incluyen la redacción de la memoria y la preparación de la presentación para la defensa del proyecto. En la Gráfica 10.1, se ha representado la duración de cada una de forma aproximada.



Gráfica 10.1: planificación etapas proyecto. Fuente: propia.

Primeramente, se estudió el funcionamiento del algoritmo AES mediante la especificación [9], propuestas anteriores a la especificación [12] y un libro redactado por los propios autores del algoritmo pocos años después de proclamarse ganadores del concurso [10].

Como se puede observar en el Gráfica 10.1, esta parte se extiende casi hasta el final de la optimización, ya que se mejoró el programa observando detenidamente función por función y estudiando cada una a fondo. Fue necesario leer y pensar varias veces cada uno de los conceptos para comprenderlo bien.

De forma paralela, me fui familiarizando con el lenguaje de programación C con el que se desarrollaría el proyecto. Para ello, se siguió un tutorial de internet referenciado en [15]. A partir de éste, se fueron realizando pequeñas modificaciones a medida que se explicaban los diferentes conceptos propios de dicho lenguaje.

A medida que se iban adquiriendo competencias en la programación con el lenguaje C y se

iba comprendiendo el funcionamiento del algoritmo de cifrado AES, se fueron realizando las diferentes mejoras. El bloque de optimización se puede dividir en: búsqueda de una primera versión por internet, compilación de dicha versión, obtención de los datos de partida, optimizaciones básicas, construcción de tablas, planteamiento nuevo.

Por último, se redactó la memoria a partir de los resultados que se iban obteniendo en cada mejora y se preparó la presentación para la defensa del proyecto.

11. Posibles mejoras

En este apartado se explican, de forma resumida, algunas posibles mejoras en la optimización del código. Se plantean alternativas tanto de la parte software como de los dispositivos necesarios para el cifrado.

Por un lado, se podría utilizar un compilador de pago, en vez del gratuito utilizado en este proyecto. Con ello, seguramente no se hubiese tenido que modificar la versión original. Asimismo, la depuración automática del programa a la hora de pasarlo a código máquina sería mejor. Además, en el intento de ajustar la optimización mediante la opción *custom* (véase el apartado 6.9), posiblemente se habrían conseguido mejorar los valores de los criterios.

Por otro lado, se podría haber estudiado el lenguaje ensamblador para ser más preciso a la hora de escribir el código o para incluir algunas líneas en ensamblador dentro del código escrito en C, principalmente en los cuellos de botella como, por ejemplo, la función *xtime*.

Además, se ha planteado la posibilidad de usar otro microcontrolador aparte que se encargase de la expansión de la clave. Así, se conseguiría reducir el tiempo de ejecución y el uso de memoria en el descifrado, ya que no se tendrían que incluir las tablas correspondientes.

Sin embargo, la transmisión de la clave extendida entre las placas se tendría que analizar detenidamente porque la información no estaría encriptada y, consecuentemente, sería vulnerable ante posibles ataques. Se tendría que estudiar la forma de que la comunicación entre placas sea segura.

Conclusiones

Las principales conclusiones que se extraen del presente proyecto son:

- La criptografía está adquiriendo cada vez más importancia en todos los ámbitos. Sin embargo, a nivel industrial, la seguridad en las comunicaciones no se ha desarrollado tanto como en otras áreas como, por ejemplo, los métodos digitales de acreditación (DNI, cuentas bancarias, etc.).

- Los principales estándares de cifrado están diseñados para que se puedan usar en el mayor número de plataformas posibles, incluidas los microcontroladores. Por lo tanto, no se trata de diseñar un nuevo algoritmo, sino adaptar y, principalmente, optimizar los ya existentes.

- Estudiando detenidamente el código del programa y centrándose en la optimización, se pueden reducir de forma considerable tanto el uso de la memoria como la velocidad de ejecución. En el caso estudiado, se ha reducido de 54 a 7,6 ms el tiempo de ejecución y el 35% en las instrucciones de Memoria de Programa.

- La optimización demuestra que la implementación de un algoritmo de cifrado de prestigio como el que nos ocupa es viable, aunque se limiten los recursos de los microcontroladores, ya de por sí escasos.

- Dado que los microcontroladores, por su naturaleza, suelen tener una tarea muy concreta, en el impacto del rendimiento debe considerarse principalmente una rutina: la de cifrado o la de descifrado.

Agradecimientos

Me gustaría agradecer en primer lugar a Manuel Moreno Eguílaz, el director de este proyecto, por ofrecer ideas interesantes, por darme la oportunidad de participar en una de ellas, por su disposición en todo momento a echarme una mano y por sus correcciones a la hora de escribir el presente informe.

También quería agradecer a mi familia por su paciencia y apoyo, especialmente a mi padre, ya que me aportó ideas en algunos puntos del proyecto y me ayudó a seguir adelante cuando me encallaba en determinados momentos a la hora de optimizar el programa.

Bibliografía

Referencias bibliográficas

[1] Velasco, J. J. (20 de Mayo de 2014). Breve historia de la criptografía. *eldiario.es*.

[2] Real Academia Española. (octubre de 2014). *Diccionario de la lengua española (23.ª ed.)*. Obtenido de http://dle.rae.es/?w=criptograf%C3%ADa

[3] Universitat de València. (s.f.). Obtenido de http://www.uv.es/rosado/courses/sid/Capitulo3_rev0.pdf

[4] *Github*. (4 de Enero de 2017). Recuperado el 03 de Marzo de 2017, de https://github.com/kokke/tiny-AES128-C

[5] avr-libc. (8 de Febrero de 2016). *<stdint.h>: Standard Integer Types*. Obtenido de http://www.nongnu.org/avr-libc/user-manual/group__avr__stdint.html

[6] IEEE; The Open Group. (2004). *The Open Group Base Specifications Issue 6*. Obtenido de http://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html

[7] Dworkin, M. (2001). *Recommendation for Block Cipher Modes of Operation.* NIST Especial Publication. Obtenido de http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[8] Microchip. (2008). *PIC18F2420/2520/4420/4520 Data Sheet.*

[9] NIST. (2001). *Announcing the Advanced encryption Standard (AES).* Processing Standards Publication 197.

[10] Daemen, J., & Rijmen, V. (2002). *The Design of Rijndael.* Berlín: Springer.

[11] R., A. (17 de Agosto de 2011). El algoritmo de encriptación AES, más vulnerable de lo que se creía. *El País*.

[12] Daemen, J., & Rijmen, V. (3 de Septiembre de 1999). *AES Proposal: Rijndael.* Obtenido de http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[13] Bassham III, L. E. (2002). *The Advanced Encryption Standard Algorithm Validation Suite (AESAVS).* NIST.

[14] PC BOX. (s.f.). Obtenido de http://www.pcbox.com/productos/inn25522/ordenador-

innobo-pandora-geforce-i5-6500

[15] tutorialspoint. (s.f.). *C tutorial*. Obtenido de https://www.tutorialspoint.com/cprogramming/

Bibliografía complementaria

[1] P. CABALLERO, Introducción a la criptografía. Segunda edición, Editorial RaMa, Textos Universitarios, Madrid, 2002.

[2] J. PASTOR, M. A. SARASA, "Criptografía digital. Fundamentos y aplicaciones", Prensas Universitarias de Zaragoza, 1998.

Anexos